UNIVERSITY OF THESSALY

DIPLOMA THESIS

Correlating Workload Behavior with Core Voltage Variability on Modern x86 Architectures with Machine Learning

---

Author:
Konstantinos KANELLIS

Supervisors:
Christos D. ANTONOPOULOS
Spyros LALIS

A thesis submitted in fulfillment of the requirements for the degree of Diploma

in the

Computer Systems Laboratory (CSL)
Department of Electrical and Computer Engineering

Volos, October 2018

"The statements of science are not of what is true and what is not true, but statements of what is known with different degrees of certainty."

Richard P. Feynman

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

Περίληψη

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Διπλωματική Εργασία

Συσχέτιση της συμπεριφοράς εφαρμογών με τη μεταβλητότητα τάσης πυρήνα σε σύγχρονες αρχιτεκτονικές **x86**, με χρήση μηχανικής μάθησης

Κωνσταντίνος Κανελλής

Οι σύγχρονες αρχιτεκτονικές x86 είναι εξοπλισμένες με έναν καινούργιο μηχανισμό δυναμικής κλιμάκωσης της τάσης και της συχνότητας του επεξεργαστή (Dynamic Voltage Frequency Scaling - DVFS), που ονομάζεται SpeedShift. Ο μηχανισμός SpeedShift προσαρμόζει αυτόματα την τάση τροφοδοσίας του πυρήνα και την συχνότητα του επεξεργαστή, μέσω υπο-βοήθειας υλικού, βάσει των απαιτήσεων της τρέχουσας εφαρμογής. Στην παρούσα διπλωματική εργασία διερευνούμε και αξιοποιούμε τον μηχανισμό αυτό, παρατηρώντας την αλληλπίδραση μιας δυναμικής και χρονικά-μεταβαλλόμενης εφαρμογής με την αρχιτεκτονική πάνω στην οποία εκτελείται. Συλλέγοντας δεδομένα από γεγονότα που συμβαίνουν στην αρχιτεκτονική (hardware events), τα οποία καταγράφουν πως η συμπεριφορά μιας εφαρμογής επηρεάζει δομικά στοιχεία της αρχιτεκτονικής, εξερευνούμε την συσχέτιση τιμών τους με των τιμών της τάσης λειτουργία πυρήνα του επεξεργαστή, για ένα ευρύ φάσμα εφαρμογών. Στην συνέχεια εφαρμόζουμε διάφορους αλγορίθμους μηχανικής μάθησης για να εκτιμήσουμε την τιμή της τάσης λειτουργίας πυρήνα του επεξεργαστή σε οποιαδήποτε χρονική στιγμή. Παρατηρούμε πως κάνοντας χρήση νευρωνικών δίκτυων, τα οποία έχουν εκπαιδευτεί με βάση τους hardware events, πετυχαίνουμε μεγάλη ακρίβεια στις προβλέψεις μας. Εν τέλει, σχεδιάζουμε ένα καινοτόμο εργαλείο που ανιχνεύει και αναλύει αλλαγές μεταξύ των φάσεων εκτέλεσης μιας εφαρμογής, λαμβάνοντας υπόψην την τάσης λειτουργίας πυρήνα του επεξεργαστή, καθώς και μιας μετρικής κλιμακωσιμότητας της εφαρμογής (π.χ. παραγωγική συχνότητα επεξεργαστή), που παρέχεται από τον ίδιο. Το εργαλείο που αναπτύξαμε καταφέρνει να συσχετίσει τις παραπάνω τιμές με το κομμάτι του κώδικα εφαρμογής που εκτελείται, παίρνοντας μετρήσεις του μετρητή προγράμματος (program counter). Δείχνουμε ότι το εργαλείο μας είναι αρκετά αποδοτικό στο να εντοπίζει τις φάσης μιας εκτελούμενης εφαρμογής, ενώ η μεθοδολογία που ακολουθείται είναι λιγότερο επεμβατική στο σύστημά μας, η οποία και προκαλεί μονο την μισή επιβάρυνση συγκρίνόμενη με το καλύτερο διαθέσιμο εργαλείο.

UNIVERSITY OF THESSALY

Abstract

Department of Electrical and Computer Engineering

Diploma Thesis

Correlating Workload Behavior with Core Voltage Variability on Modern x86 Architectures with Machine Learning

by Konstantinos KANELLIS

Modern x86 architectures are equipped with a new, hardware-assisted, Dynamic Frequency Voltage Scaling (DVFS) mechanism (i.e. `SpeedShift`). `SpeedShift` automatically adjusts the CPU core voltage and frequency values based on the application needs. In this Thesis, we explore and exploit this mechanism in an attempt to obtain useful runtime knowledge, by observing the interaction of the time-varying behavior of applications with the architecture. At first, we explore the correlation between profiled hardware events, which capture certain aspects of the application behavior, with CPU core voltage values, for a wide set of diverse workloads. Then, we leverage supervised machine learning to accurately estimate the CPU core voltage value at any time point, and we experiment with various models. We show that, by using neural networks, it is possible to achieve great accuracy, when the neural networks are trained on hardware events data. Finally, we design a novel online phase detection tool, which utilizes CPU core voltage and a workload scalability metric provided by the architecture itself (i.e. productive frequency), to detect program phase changes. The tool also exploits instruction pointer sampling in order to associate core voltage and productive frequency samples with application context. We show that our tool efficiently discovers program phases, while being less intrusive and having half the overhead compared with the state-of-the-art methodology.

Acknowledgements

First and foremost, I would like to render my warmest thanks to my supervisor Prof. Christos D. Antonopoulos for giving me the chance to work on intriguing projects that were a perfect match for my interests. His patient guidance and availability were crucial to the completion of this thesis and allowed me to experience first-hand how research is conducted. I would also wish to express my gratitude to my advisor Prof. Spyros Lalis for his valuable comments and key suggestions throughout all stages of this work.

Additionally, I am more than thankful to my friends, old and new, for all the unforgettable moments we have lived together over the past five years. This place holds a special place in my heart because of you! Moreover, I appreciate all the help from the members of the CSL lab, who provided me with a friendly and fun work environment, and they were always there when I had all sorts of questions.

Finally, I would like to thank my family for their unconditional love, understanding and unwavering belief in me throughout all those years.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In the course of the past 40 years processors have became pipelined to increase the instruction throughput, got equipped with large multilevel caches to hide the memory latency and used advanced techniques (i.e. out-of-order execution) and superscalar designs to exploit Instruction Level Parallelism. These additions, along with the significant advances in semiconductor manufacturing techniques, resulted in tremendous performance improvements, which was generally considered as the limiting barrier at that time. More recently, and in the dawn of the post-*Dennard Scaling* [13] era, power efficiency became the primary concern for computer designers. The introduction of multicore processors partially managed to bypass the performance scaling limitations, while retaining the same power budget. However, the coupling of multiple processing cores on a single die, required the implementation of additional logic to ensure safe concurrent access to shared resources (e.g. Last Level Cache).

This rapid and irregular evolution of microprocessors, driven by the need for performance at first and power efficiency later, led to the development of robust, albeit very complicated architectures. Nonetheless, high-performance workloads should take full advantage of such computing platforms, avoiding at the same time, undesirable performance degradation often caused due to the poor utilization of some part of the processor pipeline (i.e. bottleneck). In addition, Operating Systems (OS) and CPU manufacturers design and implement dynamic software and hardware optimization mechanisms, which try to adjust the processor profile to the needs of the executed workload. However, in order to reach their full potential, these mechanisms need to accurately know the state of the architecture at any point.

Unsurprisingly, different applications have entirely different impact on the utilization of each hardware component. For example, matrix multiplication that consists of many integer (or floating point) operations is limited by the number of CPU arithmetic units. On the other hand, a linear search over a large array will be limited by the bandwidth (and latency) of either the caches or the main memory. Moreover, researchers have shown that large and complex applications consisting of thousands

of lines of code, exhibit time-varying behavior [58], which is attributed to the different code portion that is being executed. This behavior produces a distinct signature at run-time, which if traced and exploited correctly can lead to opportunities for further performance and power optimizations.

In order to get an insight of the effects of their applications on the micro-architecture, programmers use software profilers. A software profiler is a tool that monitors the run-time behavior of an application by gathering interesting program data and events. Examples of popular profilers are *perf_events* [37], *PAPI* [61] and *Intel VTune* [29]. Most advanced tools (i.e. Intel Vtune) are able to even characterize the application, using for example the *Top-down Microarchitecture Analysis Method* [68] or find possible bandwidth limiting bounds in some architectural component, by applying the *Roofline model* [66].

Software profilers typically run on top of the application they want to monitor. They employ dedicated special-purpose registers, which are located in each CPU core. These registers are called hardware performance counters or Performance Monitoring Units (*PMU*). Each register can be used to store a specific information (i.e. count or address) from a wide range of low-level hardware events (e.g. branch mispredictions, cache misses, etc.). Periodically and as long as the application is running, the profiler samples these registers and stores their values. Once the application exits, the samples from these low-level raw events can then be combined to extract more useful high-level metrics (e.g. Instructions per Cycle – *IPC*).

Unfortunately, the use of performance counters restricts the characterization of the application behavior in specific hardware domains, which depend on the events chosen to be measured. Furthermore, it is not possible to record too many hardware events at the same time, as the number of PMUs in each core is limited. In addition, the inter-core contention on shared resources cannot be deciphered efficiently from single core events. One possible way to overcome the previous limitations and acquire a bird's eye view of the behavior of the executed application, is to use aggregated architectural metrics.

Modern x86 architectures, for example `Intel Skylake` (and later), are equipped with a new power management engine called `SpeedShift` [50]. SpeedShift is advertised as an autonomous hardware-assisted Dynamic Frequency Voltage Scaler (DVFS). The motivation behind this mechanism is the quicker system response to performance burst requests. Using internal metric collection and on-chip sensors, `Speedshift` adjusts the CPU frequency and voltage, without any intervention from the OS. It, also, constantly fine-tunes the voltage and frequency values trying to compromise between the energy consumption of the CPU package and the workload performance. Thus, there is an inherent relationship between the workload behavior impact on the architecture and the voltage and frequency values set by this mechanism, at any given time.

1.1 Contributions

This thesis focuses on exploring and exploiting the aforementioned relationship, in order to obtain useful runtime knowledge on the time-varying application interaction on the architecture. This knowledge can then be potentially used in conjunction with online decision-making systems in order to apply dynamic software or hardware optimizations.

The contributions from this exploration and exploitation are the following:

- We profile a large subset of `SPEC CPU2006` [26] workloads, by sampling both hardware performance events and CPU core voltage and frequency values. Then, we model those samples as time series and we explore both the linear and non-linear correlation between the two, using statistical methods.
- After having identified some correlation, we leverage supervised Machine Learning (ML) methods to make accurate estimations on the CPU core voltage value based on the values of hardware performance events. We experiment with different ML algorithms and we evaluate their performance.
- Finally, we design and implement an online program phase detection algorithm, which uses the CPU core voltage and frequency values to decide when a change in the program behavior has occurred. By comparing its accuracy and performance overhead with the state-of-the-art [54], we show that our tool accuracy is on par, while having negligible runtime overhead.

1.2 Thesis Structure

The rest of this thesis is organized as follows:

Chapter 2 provides background on the evolution of CPU power management mechanisms and introduces the `perf_events` profiling tool, which is capable of monitoring the behavior of an application using various hardware events.

Chapter 3 presents in details the modifications that we made to `perf_events`, in order to extend its functionality for unsupported architectural events.

Chapter 4 describes the methodology we following to capture the application behavior, which ultimate led to the creation of two datasets. We explain the two different approaches we followed for selecting the appropriate hardware events, along with the post-collecting data cleaning and transformation processes.

Chapter 5 explores the linear and non-linear correlation between collected hardware events and core voltage values, for each individual workload. In addition, we characterize each workload based on its core voltage behavior.

Chapter 6 focuses on estimating the core voltage value using supervised machine learning models that were trained on specific hardware events . We experiment with diverse ML models and we evaluate their performance on two datasets.

Chapter 7 introduces a novel program phase detection and classification tool that makes use of core voltage and productive frequency values, to discover phase changes. Further, we employ unsupervised learning to efficiently classify different parts of the program execution to distinct phases.

Finally, Chapter 8 concludes this thesis by discussing our key findings and by presenting some directions for future work.

Chapter 2

Background

2.1 CPU Power Management

The physical barriers encountered in the CMOS transistor manufacturing technology, marked the end of low-cost performance scaling. Thus, the research community turned its interest in finding ways to minimize the energy consumption. Nowadays, the power management is an important concern in the design of CPUs. Modern CPUs integrate numerous different mechanisms aimed at maximizing power-savings while trying to minimize performance degradation.

In this section, we present the evolution and the inner-workings of power management techniques that have been implemented on the last few generations of `Intel` processors. Then, we give an overview of the set of the latest enhancements that constitute the `SpeedShift` technology, which we aim to exploit.

2.1.1 Power consumption model

The goal of the CPU power management mechanism is to reduce the power consumption of the processor. The CPU power consumption can be split in two parts: static power and dynamic (or switching) power, as follows:

$$P_{CPU} = P_{dyn} + P_{static}$$

where P_{static} is the power required just to keep the CPU on, and P_{dyn}, which reflects the consumption due to the activity of CPUs logic gates, and depends on the running workload. Dynamic power P_{dyn} is the dominant factor in this equation as it accounts for the majority of the CPU total power consumption. The dynamic power of CMOS transistors can be approximately modeled [43] as:

$$P_{dyn} = aCV^2f$$

where C is the switching capacitance, V is the voltage, f is the frequency and a is the activity factor (i.e. average number of transistor switching events). The key takeaway here is that the CPU power consumption is proportional to the frequency, and to the square of the voltage. Thus, by lowering just the voltage it is possible to get significant power gains, without generally affecting the performance. This is not the case with frequency though, where lower values bring some power gains but often result in longer workload execution times.

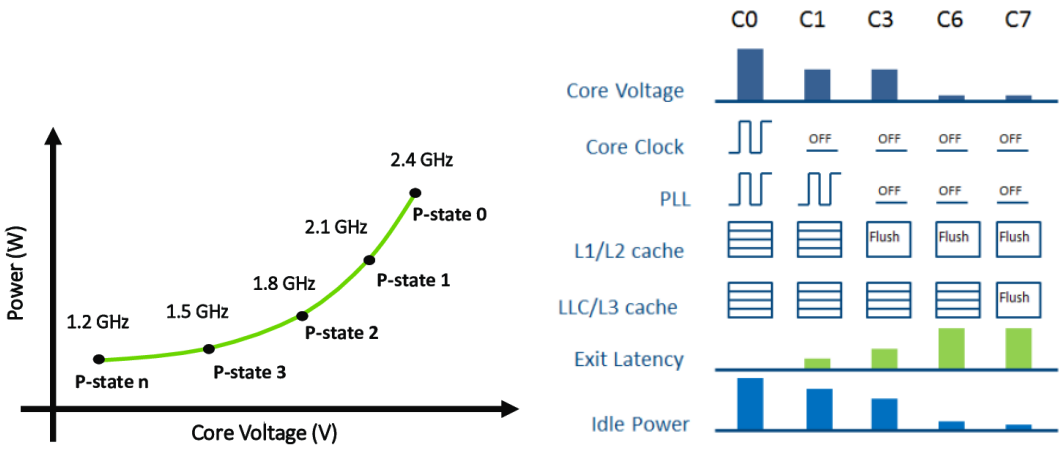
2.1.2 Dynamic Voltage-Frequency Scaling

Before the CPUs were equipped with Dynamic Voltage-Frequency Scaling (DVFS) mechanisms, they used constant voltage and frequency operation points. These adaptive mechanisms allowed for opportunistic adjustments of both voltage and frequency, especially when the full computational power of the processor is temporarily not needed. In order to ensure the stability of the CPU, however, a change on the CPU frequency is accompanied with an appropriate CPU voltage change.

The `Intel SpeedStep` technology [51] released in 2002 (with the Prescott 6 series), and `AMD Cool'n'Quiet` in 2003, were the first DVFS implementations for consumer desktop processors. These mechanisms depend on the operating system to find the optimal power configuration, based on the current system load. This is achieved using CPU utilization statistics (i.e. `APERF` & `MPERF` registers) provided by x86 platform [39]. Then, the OS applies the optimal configuration using the Advanced Configuration and Power Interface (`ACPI`), which relies on the P-states and C-states (explained below). The above process is repeated periodically (every few 10s of milliseconds), as the CPU load can change continuously (e.g. new process is spawned, process switch to sleep state, etc.). In the `Linux` kernel, this mechanism is implemented under the `cpufreq` infrastructure [65].

P-States

Depending on the current workload requirements, a CPU can operate at different discrete voltage-frequency levels called **P-States** (Performance States). Generally `P0` is the highest state (i.e. maximum performance), while `Pn` is the lowest one (i.e. maximum power-saving). Each intermediate state (e.g. `P1`, `P2` and so on) saves additional power, but at the same time adds an extra penalty to the CPU performance. Figure 2.1a shows graphically the impact of different P-states at the total power consumption of the CPU.



(A) P-States (Performance states) (B) C-States (Processor states)

FIGURE 2.1: Figure 2.1a (left) shows intuitively the impact on the processor power consumption for various P-states. Figure 2.1b (right) shows which HW components are powered-down for each C-state. [7]

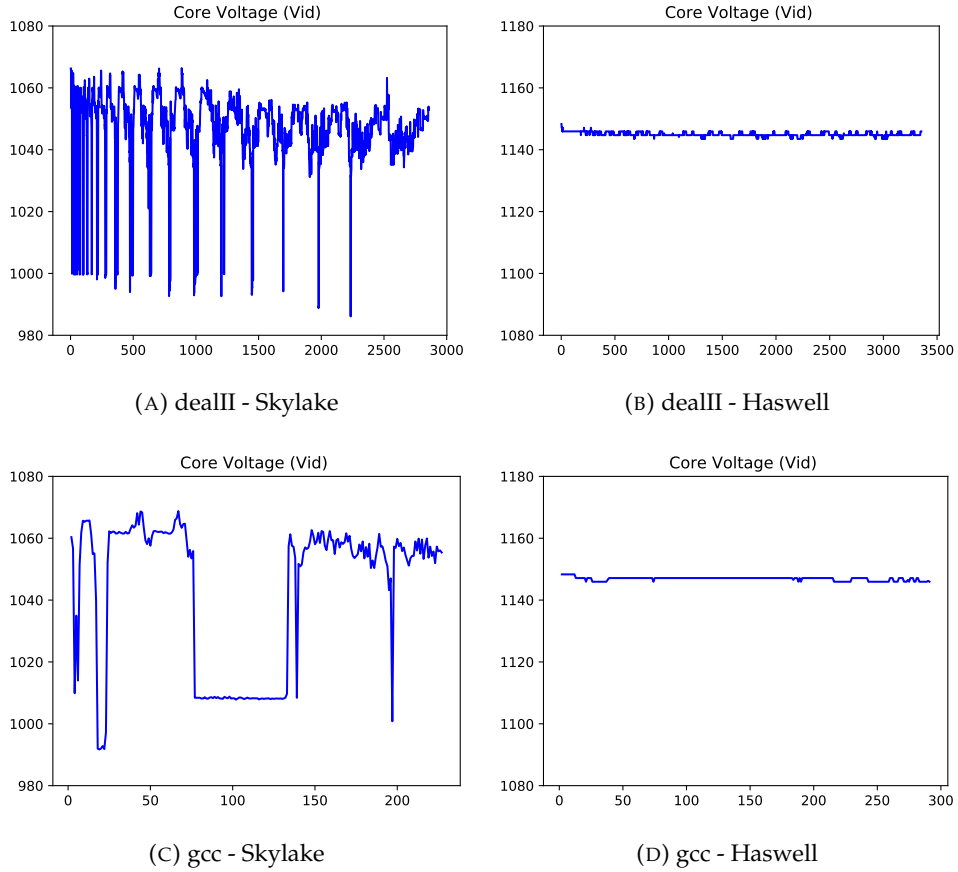
C-States

In contrast to `P-States`, which are design to optimize power consumption under workload execution, `C-states` (Processor states) are used to reduce power consumption when the processor is in idle mode (i.e. nothing is executed). At a C-state (other than `C0`), unused hardware parts are powered down to save energy seeing that no workload needs them. Each CPU core can operate on a different `C-state` which brings further power savings for single-threaded workload. Figure 2.1b illustrates which hardware components are powered-down under each `C-state`. Finally, it is worth mentioning that `P-states` are relevant only in the presence of the `C0` state, as the core clock is active only in that state.

The latest microarchitectures (i.e `SandyBridge/Haswell/Broadwell`) brought additional power and performance improvements, as these DVFS technologies were further optimized (i.e. `Enhanced Intel SpeedStep Technology`). Moreover, the `Linux` kernel support was improved with the use of the smarter `intel_pstate` governor [65].

2.1.3 Hardware-controlled P-states (HWP)

Ever since the introduction of the aforementioned power management mechanisms, it was operating system role to detect a change in the system load and properly adjust the CPU frequency-voltage values (i.e. P-state). However, when transient workloads are spawned, which are in need of a rapid performance boost, the operating system response time granularity (10s of milliseconds) is limiting. Another issue is



(A) dealII - Skylake

(B) dealII - Haswell

(C) gcc - Skylake

(D) gcc - Haswell

FIGURE 2.2: Comparison of core voltage behavior between Skylake and Haswell architectures

that the OS lacks the ability to make a direct observation of the workload microarchitectural behavior. In order to address these issues, a new set of power management technologies were implemented in the `Skylake` (and later) microprocessors.

This new technology, code-named `SpeedShift`, makes it possible to offload the frequency and voltage shifting from the operating system to a hardware micro-controller (i.e. Package Control Unit – PCU). PCU collects internal architectural statistics and monitors the power envelope of the CPU. Using the collected statistics and by computing and applying the optimal configuration every $\sim$ 1 millisecond , the processor manages to quickly adapt to bursty performance and power needs [19, 50].

SpeedShift is also capable by default, of instantaneous full-range frequency shifts. This means that it is possible for example to transition from `P4` to `P0` in a single step rather than making all the way through the `P3`, `P2` and `P1` states, until eventually reaching the desired `P0` state. The operating system, however, can still change the default configuration of the mechanism to ensure a Quality of Service (QoS). This is typically achieved by specifying the minimum and the maximum processor operating frequency [30].

Figure 2.2 compares the core voltage behavior between the recent `Skylake` architecture, and the older `Haswell` one. Two workloads were profiled: gcc and dealII, which have shown significant core voltage variability on Skylake architectures. The x-axis represents the core voltage samples, which are taken every 5 milliseconds. Note that even though the two architectures have different core voltage operating point range, the plots have the same y-axis scale. It is obvious that the core voltage on the Haswell system is almost constant, with minor variations. This is not the case on the Skylake system, as the core voltage experiences very large fluctuations throughout the execution of the workload.

2.1.4 Performance Boosting Mechanisms

Apart from the power-saving mechanisms described in the previous paragraphs, there has been some research on integrating performance-boosting mechanisms on the processors, too. One such technology is `Hyper-Threading` (HTT) which basically allowed two threads to sometimes run simultaneously on a single core [41] (i.e. Simultaneous Multithreading – SMT). Because of superscalar design of x86 processors, it is possible to improve the parallelization of the computations by issuing individual instructions in the pipeline, with the restriction that they operate on separate data. Hyper-Threading exists on almost all modern desktop and server mid to high-end CPUs.

A more advanced mechanism, commonly found in some high-end processors, is `Intel Turbo Boost Technology`, which allows some cores to run faster than the rated clock frequency for a short time [33]. Typically, this mechanism takes action only when the cores are on the `P0` state (i.e. maximum frequency) and within the power, current and thermal design specification limits. Without diving into the details, these mechanisms provide some additional P-states, called `turbo P-states`, which are used when it is considered safe.

For the scope of this thesis, we won't be focusing on these performance-oriented technologies.

2.2 `Perf_events` profiling tool

In order to monitor the processor microarchitectural behavior, we extensively use an existing performance monitoring and analyzing tool called `perf_events`, which is available in the `Linux` kernel since version `2.6.32` [37]. `Perf_events` is capable (among other things) of statistical profiling of each core or of the entire system by providing a unified interface to the low-level Performance Monitoring Units (PMU). These units may differ from one architecture to another, as their specific implementation uses special-purpose hardware registers.

`Perf_events` abstracts the implementation details and enables the hardware events reporting to the user-space. This is achieved by using a kernel component that performs and buffers the actual measurements and a user-space component, which periodically pulls and saves these measurements.

`Perf_events` can be programmed to operate in two modes:

- **Counting mode**, which enables the counting of the occurrences of certain events e.g. executed instructions, L1 caches misses, inside a user-defined interval.
- **Sampling mode**, which periodically triggers interrupts after a user-defined number of occurrences of the given event.

Since we are not interested in locating individual events but getting aggregated values, we use the counting mode. `Perf_events` provide this functionality with the `perf stat` subcommand.

Traditional hardware performance counters measure events inside a CPU core. Nevertheless, in modern `Intel` architectures there are a couple more mechanisms that provide access to package-wide measurements.

In the next paragraphs, we present a rough overview of each different type of hardware event.

2.2.1 Programmable core events

These are the most common events and usually refer to the programmable hardware performance events. These events can be monitored by properly programming the PMUs, which requires some writing to dedicated control registers. Fortunately, these low-level operations are abstracted by `perf_events`. On x86 architectures, each CPU core has a distinct set of PMUs, thus it is possible to take per-core measurements.

`Perf_events` holds a list of popular pre-defined events which usually refer to the high-level components of the processor architecture. It also presents them which user-friendly names instead of the actual (more compliacted ones). Listing 2.1 shows the output of the `perf list` subcommand. Typically, the **hardware** and **hardware cache** event types occupy a PMU, while the **Kernel PMU events** do not. However, this also depends on which hardware events are to be measured.

```
List of pre-defined events (to be used in -e):

  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
  bus-cycles                                         [Hardware event]
  cache-misses                                       [Hardware event]
  cache-references                                   [Hardware event]
  cpu-cycles OR cycles                               [Hardware event]
  instructions                                       [Hardware event]
  ref-cycles                                         [Hardware event]

  ... <omitting software events> ...

  L1-dcache-load-misses                              [Hardware cache event]
  L1-dcache-loads                                    [Hardware cache event]
  L1-dcache-stores                                   [Hardware cache event]
  L1-icache-load-misses                              [Hardware cache event]
  LLC-load-misses                                    [Hardware cache event]
  LLC-loads                                          [Hardware cache event]
  LLC-store-misses                                   [Hardware cache event]
  LLC-stores                                         [Hardware cache event]
  branch-load-misses                                 [Hardware cache event]
  branch-loads                                       [Hardware cache event]
  dTLB-load-misses                                   [Hardware cache event]
  dTLB-loads                                         [Hardware cache event]
  dTLB-store-misses                                  [Hardware cache event]
  dTLB-stores                                        [Hardware cache event]
  iTLB-load-misses                                   [Hardware cache event]
  iTLB-loads                                         [Hardware cache event]
  node-load-misses                                   [Hardware cache event]
  node-loads                                         [Hardware cache event]
  node-store-misses                                  [Hardware cache event]
  node-stores                                        [Hardware cache event]

  branch-instructions OR cpu/branch-instructions/    [Kernel PMU event]
  branch-misses OR cpu/branch-misses/                [Kernel PMU event]
  bus-cycles OR cpu/bus-cycles/                      [Kernel PMU event]
  cache-misses OR cpu/cache-misses/                  [Kernel PMU event]
  cache-references OR cpu/cache-references/          [Kernel PMU event]
  cpu-cycles OR cpu/cpu-cycles/                      [Kernel PMU event]
  instructions OR cpu/instructions/                  [Kernel PMU event]
  mem-loads OR cpu/mem-loads/                        [Kernel PMU event]
  mem-stores OR cpu/mem-stores/                      [Kernel PMU event]
  cpu/topdown-fetch-bubbles/                         [Kernel PMU event]
  cpu/topdown-recovery-bubbles/                      [Kernel PMU event]
  cpu/topdown-slots-issued/                          [Kernel PMU event]
  cpu/topdown-slots-retired/                         [Kernel PMU event]
  cpu/topdown-total-slots/                           [Kernel PMU event]
```

LISTING 2.1: Output of `perf list` subcommand

The number of the available hardware events is much higher though, and depends

Perf_events Name `Mnemonic Name`	**Event Description**
cpu/instructions `INST_RETIRED.ANY`	Instructions retired from execution
cpu/cycles `CPU_CLK_UNHALTED.THREAD`	Cycles when the core is not in halt state
cpu/ref-cycles `CPU_CLK_UNHALTED.REF_TSC`	Counts the number of reference (i.e not affected by core frequency changes) cycles when the core is not in a halt state.

TABLE 2.1: Fixed-function events in the `Skylake` architecture

on the architecture generation. On the `Skylake` architecture, the actual number of such events is approximately 220 [31]. `Perf_events` can program the PMUs to monitor any of these events by providing its (unique) event mnemonic name. Unfortunately, it is not possible to monitor all these events simultaneously, as the limited number of PMUs (per core) in `Skylake` architecture allows for the simultaneously monitoring of **4** events, or **8** if the `Hyper-Threading` technology is disabled.

2.2.2 Fixed-function events

Fixed-function events, are per core performance events that do not occupy a PMU. In contrast to programmable counters, these are special registers, which have the very specific role of measuring a certain event. In general, these registers are updated internally by the CPU at constant intervals, and thus changing the event measured is not possible.

Table 2.1 lists the fixed-function events for the `Skylake` architecture. Note the generic nature of these events, as their value can be beneficial in any kind of performance or power analysis.

2.2.3 Shared socket-wide events

The shared socket-wide events, called from now on `uncore` events, measure events that occur outside the CPU cores. The uncore PMUs (or "boxes") can be programmed to measure the number of last level cache (i.e. L3 cache) misses, cache coherence protocol snoop events and main memory (i.e. RAM) cache line requests and many more [31]. There exist various types of "boxes", which are located in different parts of the architecture to allow for the effective profiling of the subsystems. For example, a `C-box` is dedicated to the last level cache metrics, the `iMC` refers to the integrated memory controller etc. More technical details related to uncore events, can be found in [63].

Event Name / Register Name	**Event Description**
power/energy-pkg/ `MSR_PKG_ENERGY_STATUS`	The total amount of energy consumed by the whole package / chip.
power/energy-cores/ `MSR_PP0_ENERGY_STATUS`	The total amount of energy consumed only by the cores.
msr/thermal/ `IA32_THERMAL_STATUS`	Current core temperature

TABLE 2.2: Pre-defined power / thermal events by `perf_events`

2.2.4 Power-related and thermal events

In addition to the previous performance events, modern systems may provide on-CPU power and energy measurements. These are exposed via the Running Average Power Limit (`RAPL`) interface, which is available on certain `Intel` processors. Depending on the actual processor model, these might be just estimates, computed by math models, or actual power measurements, provided by on-chip sensors. RAPL events are divided among different components of the platform, to enable for finer control (e.g. package, CPU cores, DRAM controller etc.).

`Perf_events` supports the RAPL interface since mainline version `3.14` [38]. Table 2.2 lists the relevant power and thermal events, provided by `perf_events` for the `Skylake` architecture.

Chapter 3

Extending `perf_events`

As it has been discussed in section 2.2, the run-time knowledge of the micro architecture can be obtained using mechanisms that employ dedicated hardware registers (i.e. PMUs). These registers are just a small part of a larger group of architecture-specific registers called Model-Specific Registers (MSR) [32]. The access to the majority of the registers, which are used by some already-established mechanism (i.e. RAPL, PMU) has already been abstracted by the specifications of the interface. However, there are more MSRs available which are not part of some interface, yet they can be read directly to extract more information.

Reading from or writing to an MSR is handled by the `rdmsr` and `wrmsr` commands. These are privileged instructions, however, and cannot be executed directly from the user-space. Thus, we make some modifications to the kernel component of `perf_events` with the intention to include the measurement of events described below, which are not yet supported by the current versions [1].

3.1 CPU core voltage

According to Intel Software Developer's Manual [32], which lists all the MSR for each processor generation, it is possible to read the CPU core voltage on `SandyBridge` (and later) architectures. This is achieved by reading the value in the `IA32_PERF_STATUS` MSR and then extracting the value of `[32, 47]` bits. Through experimentation we discovered that this register value is updated every $\sim$ 1 millisecond.

Listing 3.1 presents a sample `C` implementation that returns the core voltage value (in millivolts). Please note, however, that the CPU (i.e. package) provides just a single MSR register for all cores. Now, whether it is the case that each core has it is own voltage regulator, and thus may work on a different voltage level from each other, or all cores share the same regulator, is not currently documented. For the scope of this thesis, it is assumed that all cores are powered by the same core voltage.

[1] These modifications were made by Panos Koutsovasilis

```
unsigned short read_core_voltage (int cpu_core) {
    unsigned long long vid = rdmsr(cpu_core, MSR_IA32_PERF_STATUS);
    vid = (vid >> 32) * 1000;
    return ((unsigned short) vid >> 13);
}
```

LISTING 3.1: Sample function implementation in C that returns the core voltage in millivolts, on SandyBridge (and later) processors

3.2 Core Frequency

The current CPU core frequency can be calculated in Skylake architectures by taking readings of the following two MSRs:

- **IA32_MPERF**, which increments with the **maximum** frequency (i.e. fixed clock rate), when the processor is in the C0 state.
- **IA32_APERF**, which increments with the **actual/current** frequency (i.e. clock rate of execution), when the processor is in the C0 state.

Assuming the processor is not idle, the $APERF/MPERF$ indicates the ratio of total cycles to constant-clock cycles (i.e. cycles that would had been executed if the processor was at the P0 state at the whole time). Thus, using the following formula:

$$FREQ_{CORE} = FREQ_{BASE} \times (\Delta\ APERF\ /\ \Delta\ MPERF)$$

it is possible to obtain the average core frequency over the last interval (as shown in [31] – paragraph 14.5.5). The CPU base frequency is a design-specific value and it is constant on the processor. So, the above formula can be translated to code by using the APERF and PFREQ values of the previous interval to calculate $\Delta\ APERF$ and $\Delta\ MPERF$, respectively.

3.3 Productive Performance

One new and very interesting metric, found only in Skylake (and later) architectures, is the so-called Productive Performance. As discussed in [31] on paragraph 14.4.5.1, it provides a quantitative metric to software of hardware's view of workload scalability. This can be defined as a rough estimation of the relationship between frequency and workload performance, to software.

The Productive Performance value can be obtained by reading the IA32_PPERF MSR. This counter is increased only in "productive" cycles, where the hardware believes

that there is real progress to instruction execution. In other words, "productive" cycles are the ones that the core does not experience any activity stalls due to some dependency (e.g. waiting data from memory). So, in simple terms, productive performance refers to the extent of stalls compared to stall-free cycles within a time window [44].

Analogous to the core frequency case, the $\Delta\ PPERF/\ \Delta\ APERF$ indicates the ratio of total cycles to the **productive** or **stall-free** ones (based on observations from the hardware itself). Having these in mind, we define a new metric, which we call `Productive Frequency`, as follows:

$$FREQ_{PROD} = FREQ_{BASE}\ \times\ (\Delta\ PPERF\ /\ \Delta\ APERF)$$

We can argue that $FREQ_{PROD}$ is the (average) lowest possible frequency value that we could have set the core frequency to in the last interval, so it didn't had a negative impact on the workload performance. We will see later on section 7.3, that the productive frequency can be exploited for detecting phases of workload behavior.

3.4 Processor states (C-states)

The `Skylake` architecture provides several MSRs on each CPU core that can be used to measure how much time a core spends in some idle-state (i.e C-state). Each core has dedicated MSRs for the C3, C6 and C7 states. At every cycle the processor core increments one of these MSRs if it currently is in any of these states. An additional MSR, named time-stamp counter (TSC) that is increased every cycles, can be used to find the residency percentage. This is achieved by dividing the C3, C6 and C7 state value by the TSC value, as the latter is invariant of the core C-state:

$$RESIDENCY_{CX} = TICKS_{CX}\ /\ TICKS_{TSC}\ \text{, where CX} \in \{C3, C6, C7\}$$

Since the architecture does not provide dedicated MSRs that count the time spent on C0 and C1 states, these are computed with the help of MPERF MSR. More specifically, the C1 residency can be computed as follows:

$$RESIDENCY_{C1} = (TICKS_{TSC} - \sum_{s \in S} TICKS_s)\ /\ TICKS_{TSC}$$

where S = $\{MPERF, C3, C6, C7\}$. Finally, the C0 residency can be computed by subtracting all the other C-state residencies:

$$RESIDENCY_{C0} = 1.0 - \sum_{Cx \in CX} RESIDENCY_{Cx}\ ,\ CX = \{C1, C3, C6, C7\}$$

Listing 3.2 shows a sample C code that calculates the C-states residencies using the method described above. A slight code addition can be observed when computing the cycle count (i.e ticks) of C1 residency. As the MSR values cannot be read simultaneously, but with a small delay, C1 ticks might end up with negative value. Therefore, an explicit check ensures that this will never happen.

```
void get_cx_residencies (int cpu_core, float* c0_perc, float* c1_perc,
    float* c3_perc, float* c6_perc, float *c7_perc) {

    unsigned long long mperf, c1, c3, c6, c7, tsc, cx_tick;

    // Read Cx-states, MPERF & Timestamp counter ticks
    mperf = rdmsr(cpu_core, IA32_MPERF);
    c3 = rdmsr(cpu_core, MSR_CORE_C3_RESIDENCY);
    c6 = rdmsr(cpu_core, MSR_CORE_C6_RESIDENCY);
    c7 = rdmsr(cpu_core, MSR_CORE_C7_RESIDENCY);
    tsc = rdmsr(cpu_core, IA32_TIME_STAMP_COUNTER);

    // Calculate C1-state ticks
    cx_ticks = c3 + c6 + c7;
    c1_ticks = (mperf + cx_ticks <= tsc) ?
                        tsc - mperf - cx_ticks : 0;
    cx_ticks += c1;

    // CX residencies
    *c1_perc = c1 / tsc;
    *c3_perc = c3 / tsc;
    *c6_perc = c6 / tsc;
    *c7_perc = c7 / tsc;
    *c0_perc = 1 - *c1_perc - *c3_perc - *c6_perc - *c7_perc;
}
```

LISTING 3.2: Sample function implementation in `C` that returns the C-states residencies percentage on `Skylake` processors

Table 3.1 summarizes the registers that were used for extending the `perf_events` capabilities. It is possible for a single MSR to have multiple copies across every core or even every thread (i.e scope). For example, in the case of an MSR with `core` scope, each core has its own dedicated register. If a CPU has 4 cores, then 4 values can be extracted for the same event. Furthermore, the values on these registers are different as each core executes instructions independently from the others.

3.5 Variable Interval Measurements

The implementation of perf_events reports the measurements of the hardware events at fixed intervals. However, in the course of our research, it was required to take measurements of MSR-related events in smaller intervals than PMU-related ones. Therefore, we modified the kernel component of perf_events to enable for variable interval measurements among different events.

Addr	Description	Scope	Register Name
0x198	Core Voltage (bits 47:32)	Package	IA32_PERF_STATUS
0xE7	Maximum Performance Frequency Clock Count	Thread	IA32_MPERF
0xE8	Actual Performance Frequency Clock Count	Thread	IA32_APERF
0x64E	Productive Performance Count	Thread	IA32_PPERF
0x10	Time Stamp Counter (TSC)	Core	IA32_TIME_STAMP_COUNTER
0x3FC	C3 Residency Counter	Core	MSR_CORE_C3_RESIDENCY
0x3FD	C6 Residency Counter	Core	MSR_CORE_C6_RESIDENCY
0x3FE	C7 Residency Counter	Core	MSR_CORE_C7_RESIDENCY

TABLE 3.1: List of registers used for extending perf_events

This change was pretty straightforward, as the actual sampling is performed inside a big while loop that is executed at fixed time points. First, we reduced the time delay between two consecutive iterations, which correspond to MSR-related events measuring. Then, in order to measure the PMU-related events, we added logic that samples these only every k-th iteration. To summarize, MSR-related events are measured in each iteration, while PMU-related ones every k iterations. By properly defining k and the fixed time interval (in milliseconds) we can support all possibilities.

Chapter 4

Capturing Workload Behavior

4.1 Workload selection

In order to achieve a comprehensive analysis of the inner-workings of SpeedShift, we use a large subset of the `SPEC CPU2006` [26] benchmarking suite. This suite consists of a wide range of CPU-intensive workloads, written in `C`, `C++` or `Fortran` programming languages, which originate from real-world applications. Each workload may perform integer or floating-point operations and stresses different part the system's processor and memory subsystem.

Table 4.1 lists the workloads from the selected SPEC CPU2006 subset along with more information regarding their application domain.

4.2 Workload Profiling

4.2.1 System Specs & Setup

The workloads are profiled on a typical Intel `Xeon` workstation. Table 4.2 contains the full hardware and software specification of this system. Please note that the designated nominal supply voltage is the maximum possible, when there is full utilization of the CPU.

Before running the workloads, a bash script is run to perform the initial setup of the system. This setup / configuration script, the code of which is shown in Listing 4.1, does the following actions:

- It disables the `NMI Watchdog`, which conflicts with `perf_events` when reading or writing the MSRs. If not disabled, it can lead to reduced accuracy.
- It loads the `msr` module to the kernel, using the `modprobe` command. This is necessary in order for the modified `perf_events` version to be able to access the aforementioned MSRs.

Name	Operation Type	Language	Application Domain
bwaves	Floating-Point	Fortran	Fluid Dynamics
bzip2	Integer	C	Compression
dealII	Floating-Point	C++	Finite Element Analysis
gamess	Floating-Point	Fortran	Quantum Chemistry
gcc	Integer	C	C Compiler
gobmk	Integer	C	AI / Go
gromacs	Floating-Point	C	Molecular Dynamics
h264ref	Integer	C	Video Compression
hmmer	Integer	C	Search Gene Sequence
lbm	Floating-Point	C	Fluid Dynamics
leslie3d	Floating-Point	Fortran	Fluid Dynamics
libquantum	Integer	C	Physics / Quantum Computing
mcf	Integer	C	Combinatorial Optimization
milc	Floating-Point	C	Quantum Chromodynamics
namd	Floating-Point	C++	Molecular Dynamics
omnetpp	Integer	C++	Discrete Event Simulation
perlbench	Integer	C	Programming Language
povray	Floating-Point	C++	Image Ray-tracing
sjeng	Integer	C	AI / Chess
soplex	Floating-Point	C++	Linear Programming Solver
sphinx3	Floating-Point	C	Speech Recognition
tonto	Floating-Point	Fortran	Quantum Chemistry
xalancbmk	Integer	C++	XML Processing
zeusmp	Floating-Point	Fortran	Physics / CFD

TABLE 4.1: Subset of `SPEC CPU2006` workloads used for profiling

- It enables the `SpeedShift` technology on each core, by setting the least significant bit of the `IA32_PM_ENABLE` MSR.
- It locks the core frequency of all 4 cores to the fixed base frequency of the processor (i.e. 3GHz) . This is achieved by writing at the `IA32_HWP_REQUEST` MSR (more details can be found in [31] on paragraph 14.4.4). The fixed frequency allows us to not worry about potential run-time changes across `P-states` but instead focusing on the voltage variation on a single one.

```
#!/bin/bash
echo 0 > /proc/sys/kernel/nmi_watchdog  # Disable the NMI watchdog
sudo modprobe msr                       # Expose MSRs on user-space
sudo wrmsr -a 0x770 0x1                 # Enable SpeedShift on each core
sudo wrmsr -a 0x774 0x19e0001e1e        # Fix the core frequency to 3.0GHz
```

LISTING 4.1: System initial pre-profiling configuration script

In addition, we disable the Turbo Boost Technology to prevent any core from briefly switching to some turbo P-state. To further limit the interference from the operating system, we also disable the internal Intel `P-state` governor. This is achieved by selecting the `acpi-cpufreq` driver, instead of the default `intel_pstate` one, which is

Parameters	Values
CPU	Xeon E3–1220 v5
# of Cores / Threads	4 / 4
CPU Base Freq.	3.00 GHz
CPU Max Turbo Freq.	3.50 GHz
L1 D-Cache	32KB / core
L1 I-Cache	32KB / core
L2 Cache	256KB / core
L3 Cache	8 MB
RAM Size	8 GB
RAM Type	DDR4 @ 2133 Mhz
Technology	14nm
Supply Voltage (V_{dd})	1.15V
Thermal Design Power	80 W
Operating System	Ubuntu 16.04
Linux Kernel Version	4.10.17

TABLE 4.2: Specifications of the `Skylake` workstation

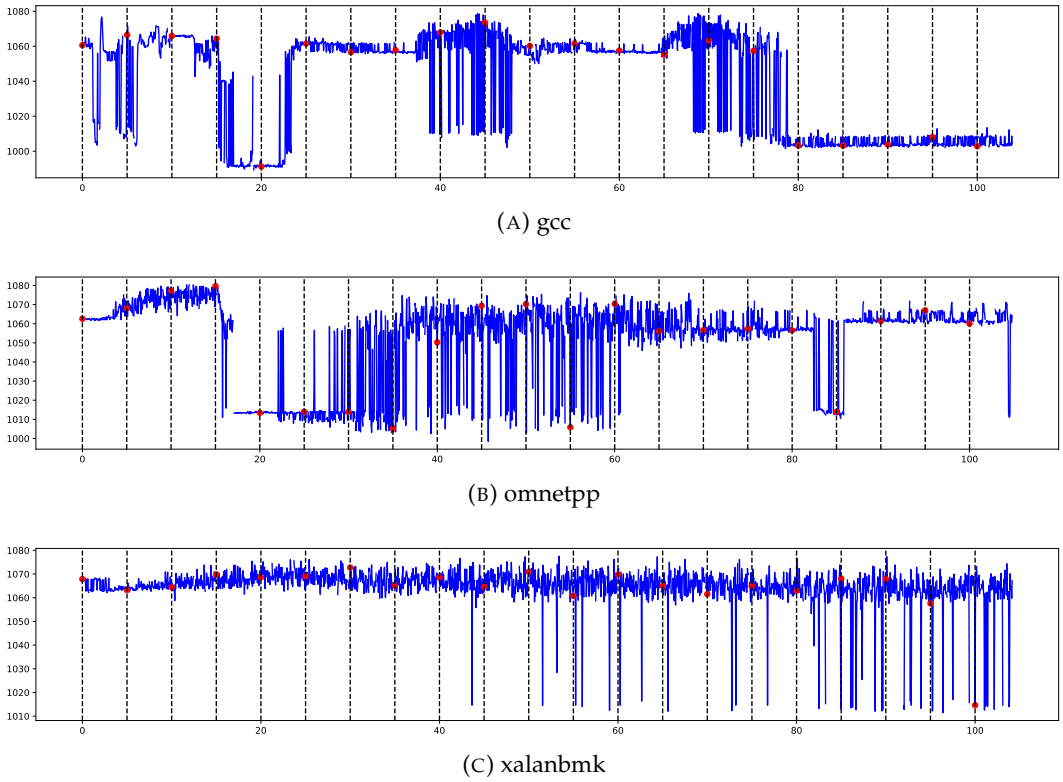
present in Linux kernel (details are available in [65]). After these steps, the full control of frequency and voltage operation points has been handed to the autonomous SpeedShift mechanism.

Finally, we launch one workload instance for every core, and we bind them to individual cores (using the `taskset` command), to be certain that they will not migrate among cores.

4.2.2 `Perf_events` Sampling Interval

The sampling rate of the events measured greatly depends on their rate of change. Figure 4.1 shows how quickly the core voltage value changes in the first half second of the execution of three representative workloads (i.e. gcc, omnetpp, xalanbmk). Samples are taken every **5 milliseconds** and the red dots indicate samples that are 100ms apart. Notice that while the "red" samples of xalanbmk workload are representative of overall core voltage behavior, this is not true for some parts of gcc and omnetpp execution.

`Perf_events` supports the sampling of aggregated hardware events that are measured by the PMUs at different time intervals. The user can specify the desired time interval depending on her needs. In our case, we are interested in observing the relationship between hardware events and core voltage. As it has been shown though, core voltage may flunctuate very quickly. Therefore, the lowest possible sampling rate should be preferred, which is found to be **100 ms**. Lower values, produce a perf_events warning message, stating that significant overhead might be introduced,



(A) gcc

(B) omnetpp

(C) xalanbmk

FIGURE 4.1: Comparison of core voltage behavior for three representative SPEC2006 workloads

mainly due to the time spent for the configuration of PMUs. Thus, we stay with 100 ms, which is applied for the core, uncore and fixed-function events / counters.

In contrast to PMU hardware events, the sampling of events that are measured using dedicated MSRs has no real overhead. Their values are updated internally by the CPU in constant intervals, regardless of the actions of the operating system. Since the only step in gathering these measurements is just reading an MSR (using the `rdmsr` instruction), the overhead is minimal as this call is done inside the kernel space (as described in Chapter 3) and does not require any expensive context switches. The selected sampling interval for these events is set to **5 ms**, and is applied for the power events, the core voltage, the core and productive frequency and the C-states residency events.

To summarize, for **each** PMU-related sample we obtain **20** MSR-related samples.

4.3 Data Collection, Cleaning & Transformation

In our evaluation system, we are able to measure just **8 core hardware events simultaneously**, as this is the number of PMUs per core. This is a very limiting if we recall that the number of available hardware events is over 220. Keeping these in

mind, there are two approaches here: either we explicitly choose some high-level metrics that have previously been successfully used in other domains or we somehow manage to get measurements for the majority of events and then, by leveraging data mining and statistical methods, select which best describe the core voltage behavior. We experiment with both, as we create two separate datasets. The exact methodology followed is described in the next paragraphs.

4.3.1 High-Level Metrics (HLM) dataset

A lot of work has been done in power consumption estimation of modern processors, using hardware performance counters. Even though, one could argue that estimating power is the same as estimating voltage, we saw that this is not true, as voltage is an instantaneous event, in contrast to power which is an aggregated one. Nevertheless, researchers have suggested a variety of hardware events that span across multiple components of the architecture, as presented in [59] and [10].

A more thorough investigation of hardware events has been presented by the Top-Down Microarchitectural Analysis (TDMA) [68]. TDMA is a practical method for quickly characterizing a workload based on the performance bottlenecks, caused by stalls in the architectural level. It employs a hierarchical organization of event-based metrics, which measure the state of the micro-architectural entire spectrum. The first level of TDMA hierarchy consists of the following metrics:

- **Front-end Bound**, which denotes when the front-end part of the architecture undersupplies the back-end. The front-end contains the branch predictor and the instruction fetcher and decoder units.
- **Bad Speculation Bound**, which refers to time wasted due to incorrect speculations. The wasted time is due to branch miss-predictions and machine clears (i.e. pipeline flushes).
- **Retiring Bound**, which reflects the retiring rate of micro-ops / instructions from the arithmetic and floating-point units (both scalar and vectorized).
- **Back-end Bound**, which measures the stall count, which occurred due to lack of required resources. This can be further split into memory bound, where execution is stalled due to data-cache misses, and core bound, where the stalling reason is the overload of the Arithmetic Logic Unit (ALU).

Although TDMA aims at improving performance, which is not related directly to power behavior, there is close resemblance on the hardware events chosen by both studies. However, this events list does not explicitly include events that are targeting the behavior of both caches and main memory. Hence, to fill this gap we include

Event Name	Event Description
Core Events	
IDQ_UOPS_NOT_DELIVERED .CORE	Uops not delivered to Resource Allocation Table (RAT) per thread when backend is not stalled
UOPS_ISSUED.ANY	Uops that RAT issues to Reservation Station (RS)
UOPS_RETIRED.RETIRE_SLOTS	Counts the retirement slots used
INT_MISC.RECOVERY_CYCLES	Core cycles the allocator was stalled due to recovery from earlier clear event for this thread (e.g. misprediction or memory nuke)
RESOURCE_STALLS.ANY	Counts resource-related stall cycles
MEM_LOAD_RETIRED.L1_HIT	Retired load instructions with L1 cache hits as data sources
MEM_LOAD_RETIRED.L2_HIT	Retired load instructions with L2 cache hits as data sources
MEM_INST_RETIRED.ALL_LOADS	All retired load instructions
Uncore Events	
UNC_CBO_XSNP_RESPONSE .MISS_EVICTION	A cross-core snoop resulted from L3 Eviction which misses in some processor core
UNC_CBO_CACHE_LOOKUP .ANY_MESI	L3 Lookup any request that access cache and found line in MESI-state
UNC_ARB_TRK_OCCUPANCY.ALL	Number of all Core entries outstanding for the memory controller
UNC_ARB_TRK_REQUESTS.ALL	Total number of Core outgoing entries allocated. Accounts for Coherent and non-coherent traffic
Fixed Events	
cpu/cycles	Core cycles when the thread is not in halt state
cpu/instructions	Instructions retired from execution
Power Events	
power/energy-pkg/	The total amount of energy consumed by the whole package/chip
power/energy-cores	The total amount of energy consumed only by the cores
Miscellaneous	
msr/vid	CPU Core voltage (in millivolts)
msr/freq	CPU Core frequency (in MHz)
msr/productive_freq	Productive Frequency (in MHz)
msr/c0_residency	CPU Residency Percentage in C0 state

TABLE 4.3: Total hardware events measured for the HLM dataset

specific events that primarily focus on the power footprint of the different cache levels (i.e. L1, L2, LLC). Table 4.3 lists the hardware events measured by `perf_events`, which are used for the construction of **HLM dataset**.

4.3.2 All-Metrics (AM) Dataset

Although the gathering of all hardware events is not feasible in as single run, many individual profiling runs can be performed. In order to profile $\sim$ 220 events, 28 distinct runs were required. The workload execution times were varying slightly, as different hardware event sets introduce different overhead. To adjust the unequal number of samples, a number of the last samples was dropped for each run so that each run had the same number of samples.

Multiple runs means that there are multiple values of each MSR event, as these are profiled for each run. Still, for constructing the final dataset, a single and hopefully representative value, should be selected. One option is to take the average of all values profiled. However, this average is a synthetic value, since it was not obtained from a direct measurement. Thus, we choose the median MSR value across the all runs, as it is the result of a real run.

4.4 Data Cleaning

Once the profiling data are collected and exported to `.csv` files by perf_events, the data cleaning process is taking place to ensure that the acquired data are relevant and can be included in the final dataset. Data cleaning (or data cleansing) usually refers to removing incorrect or inaccurate samples obtained during the collection phase [67]. In our case, the following cleaning steps are performed:

- The samples with a **C0-state residency** value below 80% are discarded. The rationale behind this decision is that at this time interval, the processor is not "busy enough", and thus the aggregated hardware events values obtained cannot be directly compared against other intervals. Depending on the workload, this discards from 0.5% to 2% of the total samples collected.
- The samples having a **core frequency** value outside of the $2.95 - 3.05$ GHz range are dropped as well. It has been observed that on some workloads there exist some sudden peaks on the core frequency value. As frequency changes may impact the core voltage as well, we want to make sure that our samples are consistent to the same voltage range. This step drops less that 0.5% of the samples for every workload.

The above steps are applied to both **HLM** and **AM** dataset samples.

Metric Name	Formula
Core Metrics	
Frontend Bound	FetchBubbles / TotalSlots
Bad Speculation	(SlotsIssued – SlotsRetired + Recovery-Bubbles) / TotalSlots
Retiring	SlotsRetired / TotalSlots
Backend Bound	1 – (Frontend Bound + Bad Speculation + Retiring)
Instructions Per Cycle	Instr / Cycles
L1 Cache Hit Percentage	MemHitL1 / MemLoadInstr
L2 Cache Hit Percentage	MemHitL2 / MemLoadInstr
Memory Loads Fraction	MemLoadInstr / Instr
Uncore Metrics	
Memory Miss Eviction Perc	MemMissEviction / TotalSlots
Memory Cache Lookups Perc	MemCacheLookups / TotalSlots
Memory Request Latency	MemOccupancy / MemRequests

TABLE 4.4: Tranformation of gathered events for the **HLM dataset**

4.5 Data Transforming

4.5.1 PMU-related events

The collected PMU data of the **HLM dataset** are transformed based on the principles presented by the TPDA. The transfomation process includes adding, subtracting and diving between sampled hardware event values, to obtain a more meaningful metric. For the above-mentioned high-level metrics (i.e. Front-end & Back-end Bound, Bad Speculation and Retiring), simple formulas exist that do the conversion. Our own memory-related event values can also combined using formulas, to compute the percentage of L1, L2 and L3 cache hits, as well as the data loads fraction to all instruction. Finally, the main memory request latency time is computed with the use of two uncore hardware events. Table 4.4 presents the transformation formulas for each metric.

Since the hardware events of the **AM dataset** are so diverse, it is impossible to derive separate formulas. Nevertheless, all the aggregated event values increment with respect to core cycles. Thus, we divide all collected events with the core cycles.

4.5.2 MSR-related events

The situation is different for the events read directly from MSRs, which already have a physical meaning. Therefore, these events are neither converted nor combined together. However, recall that for each PMU-related event value we sample 20 different values from MSRs. Even though these values can help us visualize better their behavior, yet for data analysis purposes single values shall be selected. Again,

as with the previous case of multiple MSR values due to multiple runs on the **AM** dataset, we chose the median of the 20 samples (i.e. across the 100ms interval) for the same reasons.

Chapter 5

Data Analysis

This section shows graphically a small, yet sufficient subset of the collected data, to further establish the motivation behind this thesis. Furthermore, a correlation analysis is made, using the collected samples. This analysis focuses on the relationship between hardware performance events and the CPU core voltage. For both cases, the **HLM dataset** is used, as the transformed high-level metrics decently capture the individual behavior of most microarchitectural components.

5.1 Data Visualization

Although the volume of the collected data is huge (i.e 24 workloads and $\sim$ 3.5 hours of profiling the **HLM** dataset), and thus it is impossible to plot everything, we can get a good insight with respect to the rest of the data by visualizing few workloads that have interesting behavior.

5.1.1 Workload Events Behavior

Figure 5.1 shows the majority of hardware counters for `bzip2` workload, as they were sampled in a single run. Notice how the core voltage fluctuates between high (around 1070 mV) and low values (around 990 mV), but at the same time exhibits large periods of stability. The voltage changes between the high and low values are instantaneous, as the processor tries to adapt to the workload needs. This adaption (and possible speculation) process of the underlying mechanism, can also be seen from the random few voltage spikes.

Another thing to look at, is the behavior of the hardware events over time. Apart from the Productive Frequency behavior, which despite the large spikes at the voltage droops points, does not look alike the core voltage, all events show some similarity. For example, the `RESOURCE_STALLS.ANY` and `BE_BOUND` event values, display impressive similarity with the core voltage (and with each other). Furthermore, similarity can be also observed between other pairs, such as `FE_BOUND` and `BAD_SPEC`

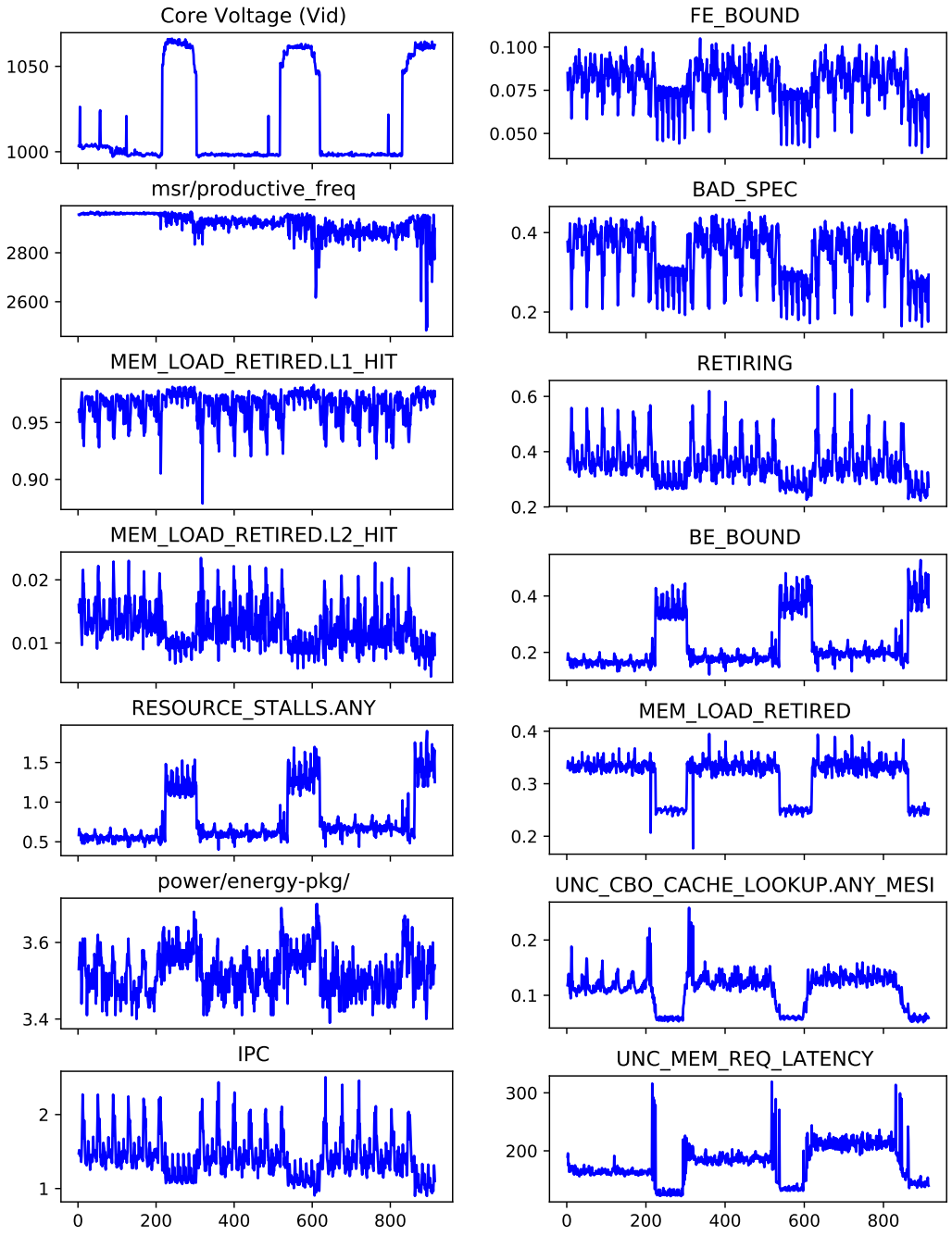


FIGURE 5.1: Time-varying behavior of `bzip2` hardware events

or `IPC` and `MEM_LOAD_RETIRED.L1_HIT`. However, the fast fluctuations observed in these events do not seem to reflect the core voltage behavior, which is for the most part stable. One might assume that some events (especially those with intense fluctuations) cancel each other and thus have no real impact on the core voltage value, though there is no practical way of proving this statement.

5.1.2 Core Voltage Behavior

Unfortunately, due to the diversity of the workloads, the above observations cannot be generalized to every application. Each workload has a different impact on the architecture and thus different hardware event footprints. This can be seen clearly in Figure 5.2, where the core voltage variability for each workload is shown. The y-axis is the core voltage value while the x-axis is the sample index. Remember that samples are $\sim$ 100 milliseconds apart and that execution times differ.

On our system, core voltage values range from $\sim$ 980 mV to $\sim$ 1080 mV at the fixed 3.0GHz frequency. However, apart from `bwaves`, `bzip2`, `dealII`, `gamess`, `gcc`, `milc`, `tonto` and `xalanbmk` which exploit the full voltage range throughout their run, other workloads tend to not have intense fluctuations.

Table 5.1 shows the workload characterization based on the rapidness and the intensity observed by the core voltage variability. The majority of the workloads, can be assigned to a single category, even if its behavior changes over time. Nevertheless, `gobmk` and `omnetpp` have been assigned to two categories, as they exhibit notable variation on separate parts of their entire execution.

	Big	**Small**
Rapid	bwaves, dealII, gamess, namd, omnetpp, tonto	gobmk, gromacs, lbm, omnetpp, soplex, xalanbmk, zeusmp
Slow	bzip2, gcc, gobmk, milc	h264ref, hmmer, leslie3d, libquantum, mcf, perlbench, povray, sjeng, sphinx3

TABLE 5.1: Characterization of workloads based on the rapidness and intensity of core voltage fluctuations

Moreover, another characterization of the workloads can be made, regarding the periodic behavior of core voltage values. A behavior is said to be periodic, if it is repeating itself, on a regular (or variable) interval. Of all the workloads, `bzip2`, `milc`, `tonto` and `gamess` demonstrate constant periodic behavior for the whole run. Other workloads exhibit periodicity in some part of the execution, such as `bwaves` (i.e. in the beginning) and `xalanbmk` (i.e. before the voltage droop near the end). Finally, `deallII`'s behavior is very interesting, as it is periodic but with an increasingly larger interval.

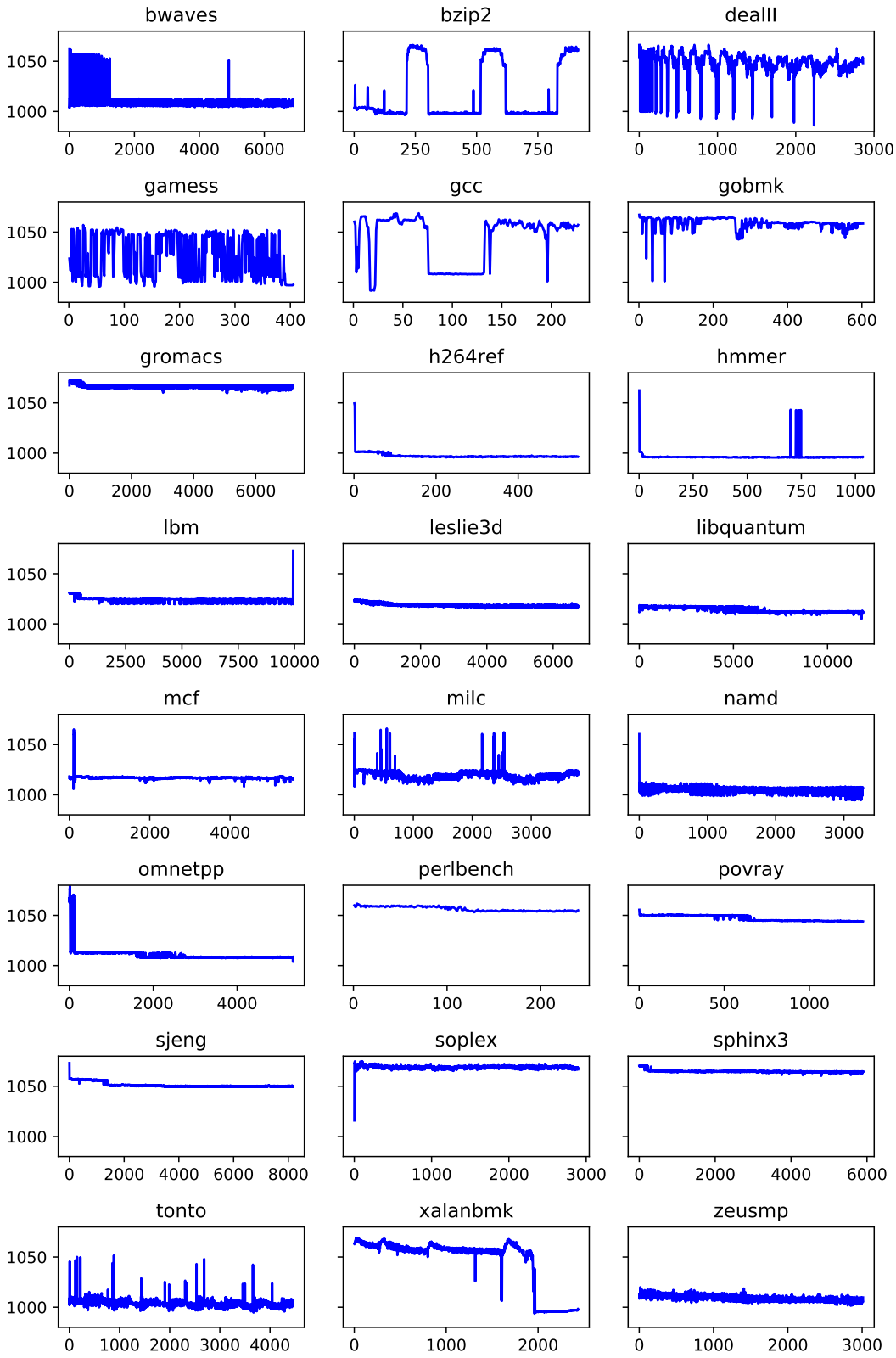


FIGURE 5.2: Core voltage variability for each workload, over time

5.2 Correlation Analysis

Correlation analysis is a set of statistical methods that quantify the strength of association between two variables over time. It can also measure, depending of the method, and the direction of this association (i.e. positive or negative). This statistical technique helps the researchers to establish if there is any perceptible relationship between the variables, and if so, potentially exploit it in some practical way. It is worth noting, however, that correlation analysis alone is not sufficient to prove a causal relationship (i.e. "correlation does not imply causation" argument).

In simple words, if a correlation is found between two variables, then when the first variable experiences some change in its values, then there should be a respective change in the values of the second variable, over a certain time period. In order to measure and explain the potential existence of a bivariate correlation, several correlation metrics (also known as **correlation coefficients**) are used. These coefficients indicate the strength and the direction of the relationship, using a single value between -1 and $+1$.

The strongest the correlation, the closer the coefficient value gets to ± 1. If there is weak correlation, the coefficient value would be closer to 0. A value of 0 indicates no correlation, which means that the bivariate values fluctuate independently of each other. In addition, the direction of the relationship can be positive or negative:

- **Positive** correlation exists if the first variable increases simultaneously with the other, and vice-versa (i.e. "$+$" sign).
- **Negative** correlation exists if the first variable increases, while at the same time the other decreases, or the opposite (i.e. "$-$" sign).

Summarizing, a coefficient value of $+1$ indicates a perfect positive degree of association between the two variables, while a value -1 a perfect negative one. Furthermore, an association can be linear or non-linear (or monotonic) depending on the rate of change of the values. In the following subsections, we explore both possibilities.

5.2.1 Exploring Linear Correlation

Pearson's Product-Moment Coefficient

Pearson's product-moment coefficient, also known as `Pearson's` r for samples, is the single most popular measure of **linear** correlation between two variables [53]. For a population the Pearson's coefficient is defined as the covariance of the two variables divided by the product of their standard deviation. However, for sampled variables the following formula is used:

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{(n-1)s_x s_y} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

where s_x and s_y are the sample standard deviations, x_i and y_i are the $i - th$ sampled points, $\bar{x}$ and $\bar{y}$ are the sample means and n is the sample size. Being a parametric statistical value, Pearson's coefficient can be vulnerable to outliers. In this case, the resulting value can be misleading [14]. However, due to the dataset pre-processing steps, mentioned in section 4.4, this is less likely to happen in our case.

Core Voltage - Hardware Events Correlation

Figure 5.3 shows the calculated Pearson correlation coefficient, between each **hardware event** and the **core voltage**, for every workload. It is observed that different workloads bring different correlation values.

We can make a workload characterization, depending on the Pearson's coefficient values, as shown in Table 5.2.

Correlation Strength	Workloads
Strong	bwaves, bzip2, h264ref, omnetpp, xalanbmk
Moderate	dealII, gamess, gcc, gobmk, namd, povray, zeusmp
Weak	gromacs, hmmer, lbm, milc, perlbench, sphinx3, tonto, sjeng
Zero	leslie3d, libquantum, mcf, soplex

TABLE 5.2: Workload Characterization based on the Core Voltage–Performance Events Linear Correlation

`Bwaves`, `bzip2` and `xalanbmk` display really strong linear correlation with most of the hardware events. However, the majority of the workloads, exhibit moderate to weak correlation with some events. It is worth noting that there are no dominant events (i.e. events that relate to core voltage for every workload). In addition, we think it is surprising that the same hardware event can have both positive and negative correlation to core voltage, depending on the profiled workload. This is the case for all events measured in the **HLM** dataset.

Some workloads exhibit (almost) zero correlation with every hardware event. This may seem concerning at first. However, as it was shown previously in Figure 5.2, these workloads have very constant core voltage signatures. Thus, it is really hard to detect any correlation anyways.

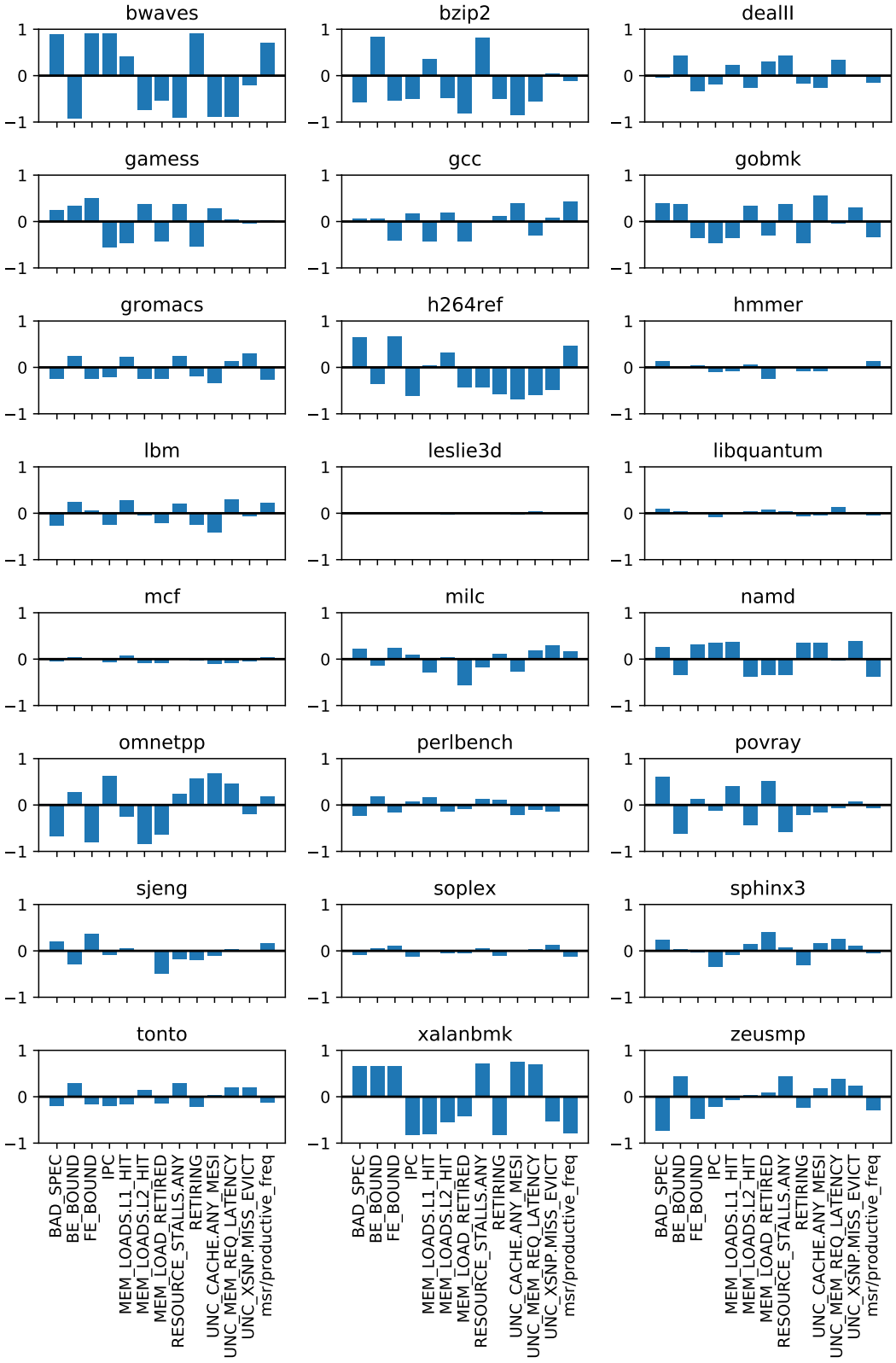


FIGURE 5.3: Pearsons's r correlation coefficient for each hardware event and workload

5.2.2 Exploring Non-Linear correlation

Spearman's Rank Correlation Coefficient

In contrast to the parametric Pearson's r, in this section a non-parametric approach is presented. The **Spearman's rank correlation coefficient**, commonly referred to as Spearmans's rho r_s, assesses statistical association based on the ranks of the data [64]. That is, it measures the statistical dependency between the rankings of two variables. By avoiding measuring directly the linear relationship between the variables, and relying instead on assessing the ranking dependence, it eventually finds how well the variables relationship can be described using a monotonic function.

At first, the x_i and y_i samples are converted to ranks $rg\ x_i$ and $rg\ y_i$ respectively. Then, by using the Pearson's coefficient definition, the correlation between the ranked variables is computed with the following formula:

$$r_s = \frac{cov(rg_X, rg_Y)}{\sigma_{rg_X}\sigma_{rg_Y}}$$

where $cov(rg_X, rg_Y)$ is the covariance of the rank variables and σ_{rg_X}, σ_{rg_Y} is the standard deviation of rg_X and rg_Y respectively. In the special case where all n ranks are distinct integers, the formula is simplified to:

$$r_s = 1 - \frac{6\sum d_i^2}{n(n^2-1)} = 1 - \frac{6\sum (rg(X_i) - rg(Y_i))}{n(n^2-1)}$$

Using the Spearman's rho coefficient, both the strength and the direction of non-linear relationships can be measured. This leads, however, to lower correlation coefficient values [16].

Core Voltage - Hardware Events Correlation

Figure 5.4 shows the calculated Spearman's correlation coefficient, between **hardware event** and **core voltage** samples, for every workload. We observe the following:

- Although, many workloads such as `bwaves`, `xalanbmk`, `omnetpp`, `h264ref` and `bzip2` had very strong linear correlation on some events, the strength of the non-linear correlation is pretty weak. In most of these workloads, the core voltage fluctuates significantly but not very often.
- Some workloads, such as `gamess`, `gobmk` and `povray` which experience fluctuations of constant range (albeit not very large), maintain almost the same correlation strength and direction.

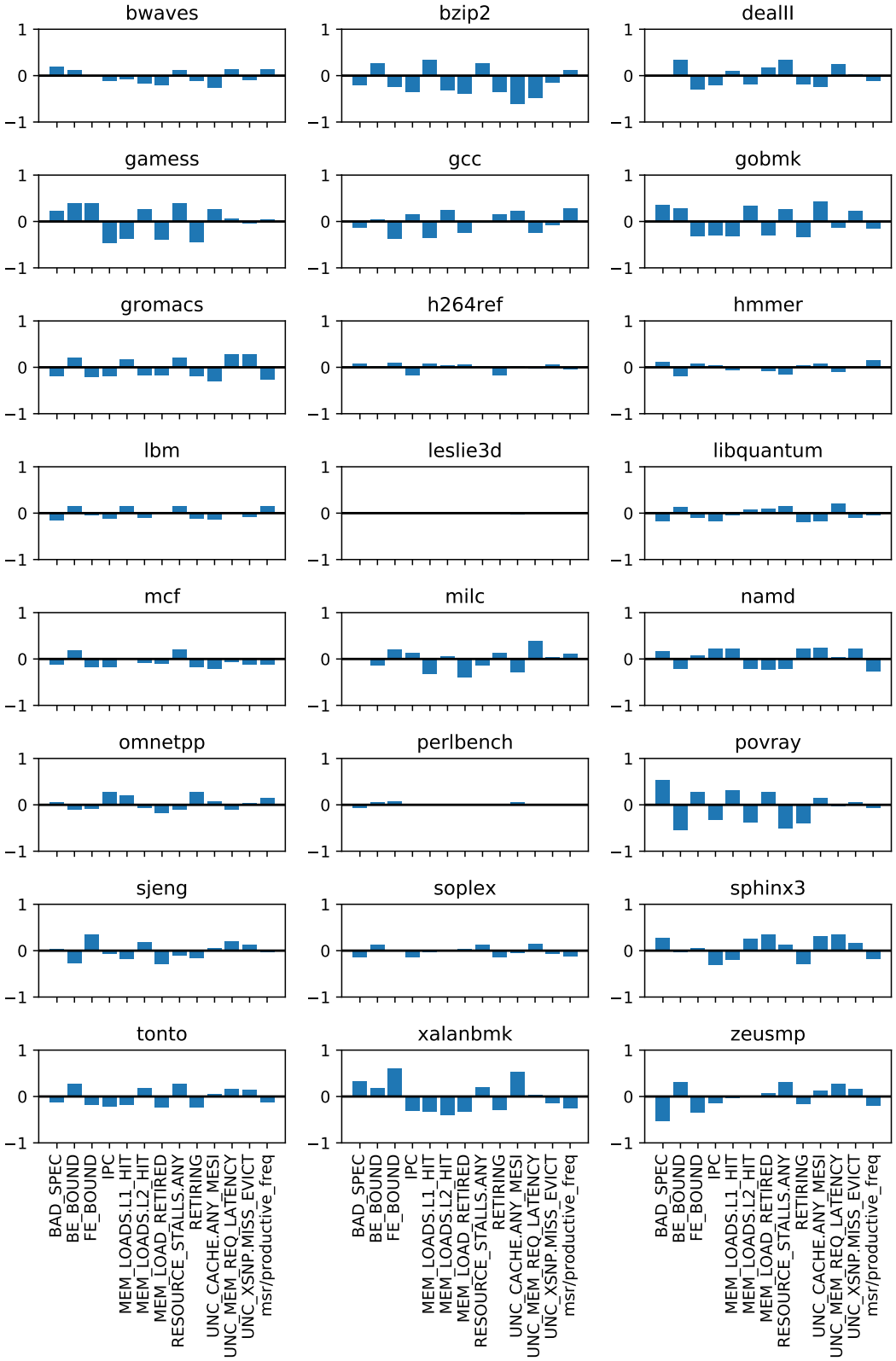


FIGURE 5.4: Spearman's r_s correlation coefficient for each hardware event and workload

- For the rest of the workloads, the Spearman's coefficient value is smaller (in absolute value) than the Pearson's one. In few cases, e.g. `libquantum`, `soplex` `mcf`, the opposite seems to happen, though these values are almost equal to zero anyways.

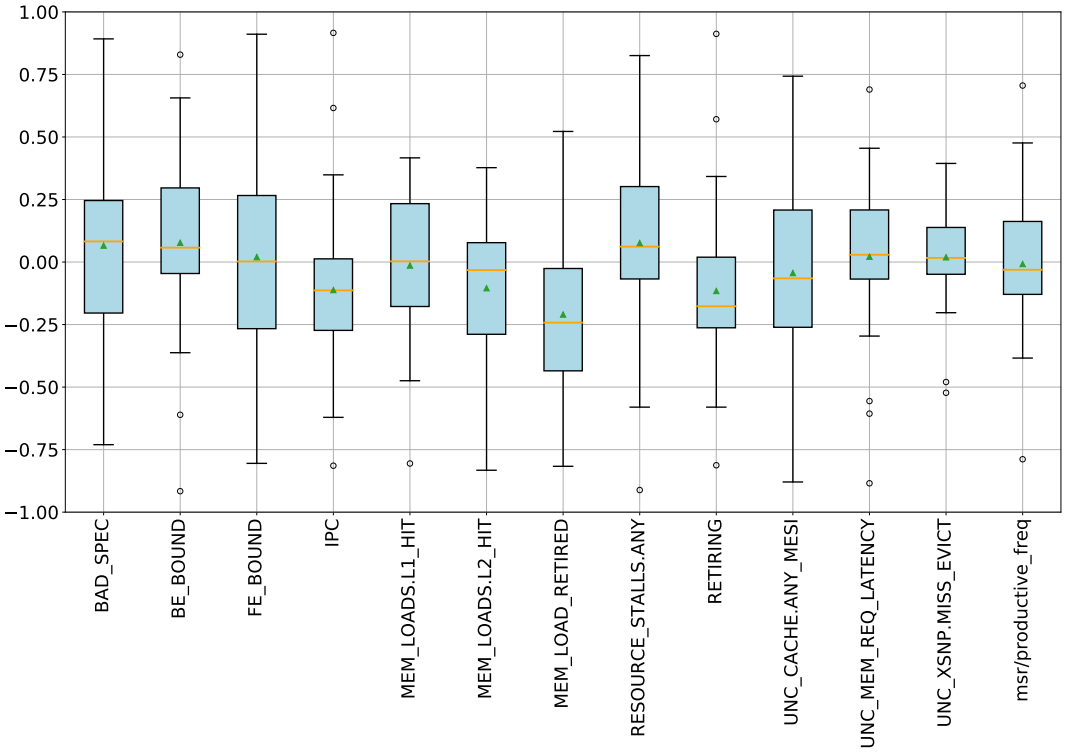
5.2.3 Cross-workload Correlation Coefficients Distribution

Boxplots are a graphical method for presenting the distribution of numerical data, through their quartiles. The data inside the second (i.e. Q_2) and the third (i.e. Q_3) quartile are represented with a rectangle, and the ones inside the lower (i.e. Q_1) and upper (i.e. Q_4) quartiles are shown with vertically lines (whiskers). Any samples located outside of the lower and upper quartiles (i.e. outliers) are plotted as individual points. The horizontal (orange in our plots) line and the triangular (green) point, which are placed inside the rectangle denote the median and the mean value of the distribution, respectively.

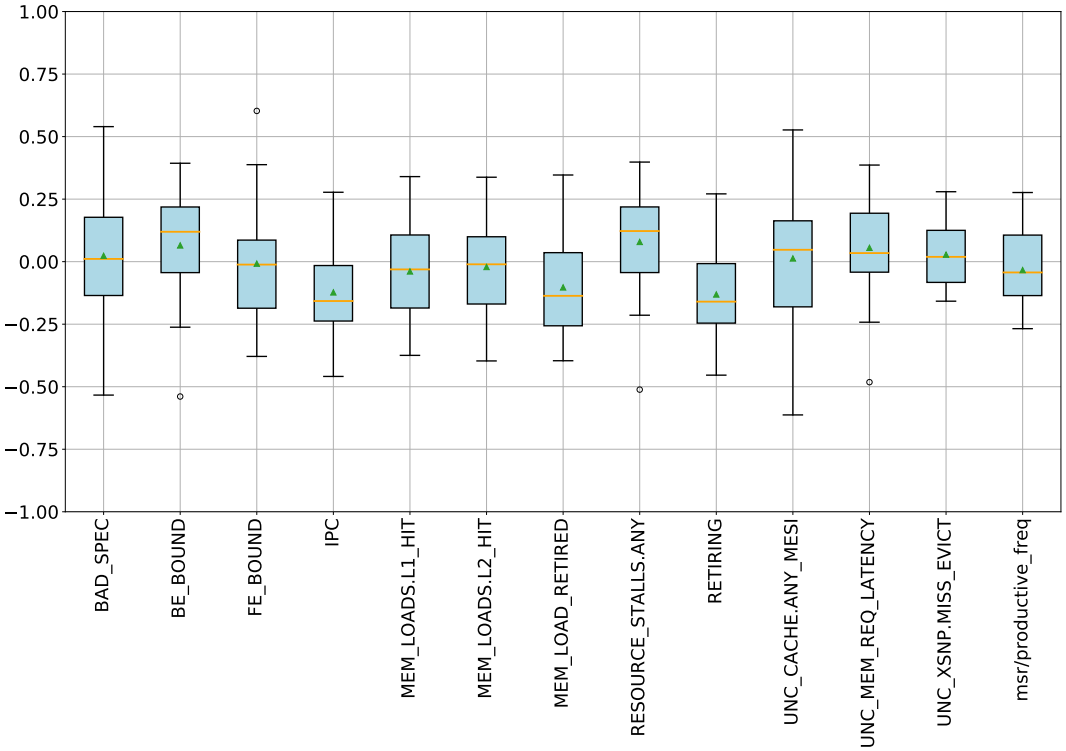
Figure 5.5 shows the boxplots for the Pearson's r and the Spearman's r_s coefficient distribution for the **HLM** dataset, across all workloads. We make the following observations:

- Person's r distribution exhibits larger variance than the Spearman's one, while there is no case of strong correlation for the latter. Furthermore, there are significantly less outliers in the Spearman's plot, which translates to fewer occasions of moderate correlation. Thus, the relationship between the hardware events and the core voltage seems to be more linear in nature, than non-linear.
- In both boxplots, the coefficient distributions have medians (and means) close to zero, with the exception of `IPC`, `MEM_LOAD_RETIRED` and `RETIRING` hardware events. In addition, the Q_2 and Q_3 quartiles of the distributions are concentrated between -0.25 and 0.25. This means that on average, there is a weak correlation between each hardware event and the core voltage. Nevertheless, there are a few cases of moderate (or even strong) correlation, especially for the Pearson's r.
- The coefficient distributions are evenly spread around the zero, in most cases. Therefore, the correlation direction can be either positive or negative, depending on the selected workload. This poses a challenge for estimating the core voltage of a single workload, if there is no knowledge of its behavior.

In the next chapter, we will see how supervised machine learning models cope with estimating the core voltage of a workload.



(A) Pearson's Product-Moment Coefficient



(B) Spearman's Rank Correlation Coefficient

FIGURE 5.5: Boxplots of the Linear (top) and Non-Linear (bottom) Correlation Coefficient Distribution

Chapter 6

Estimating CPU Core Voltage

In Chapter 5, we explored the relationship between individual hardware events with core voltage, using a statistical method (i.e. correlation analysis). Having observed weak correlation in most workloads, and strong in some, we extend our analysis by using multiple variables in conjunction, with the end goal of estimating the CPU core voltage at every point in time. For that purpose, various statistical and machine learning (ML) techniques are employed, such as linear regression, random forests along with the more modern artificial neural networks (ANN). We pay particular attention to properly train the models, by pre-processing data before feeding them into our models. Finally, the predictive power of the trained models is also validated using cross-validated techniques.

6.1 Supervised Machine Learning Algorithms

Based on our previous workload profiling and the data we gathered, which contain both the hardware events and the core voltage for each point in time, it is possible to leverage supervised learning techniques to create prediction models. Supervised learning is defined as the task of learning a function that maps an input to an output, based on input-output pairs [42]. Each such pair (or sample) consists of many input values (also known as **features**, which are typically formulated as a vector, and a single output value, which is the expected / correct target value. In our case, the core voltage is the output value and the rest of the hardware events can be used as inputs.

In order for the supervised models to learn that function, the available data are split in two sets: the **training set** and the **test / validation** set. During the training phase the labelled training set (i.e. input-output pairs) is given as input to the model, which analyses it and eventually infers a mapping function. This same function is then used by the model to predict the output values for the input vectors of the unseen, validation set. Hopefully, if the learned function fits the data "reasonably" well, the model can give acceptable output value predictions (i.e. close to the correct

ones) for unseen input values. Researchers refer to this last step as model validation and is explained in detail later on.

Since our prediction value is a continuous value, we use regression models. Furthermore, due to the large number of hardware events on our datasets, we present here only models that can cope with high-dimensional input vectors. This includes models that either have the inherent property of selecting the features which best contribute to their accuracy (e.g. decision trees), or models that can be combined with efficient techniques that implicitly eliminate the use of all features (e.g. L2 regularization with ANN).

Although there are plenty of dimensionality reduction techniques, such as the Principal Component Analysis (PCA) [22] along with its variations, none of these methods gave better accuracy results in our experiments, than the ones presented here. The same holds as well for the offline feature selection methods which are based on statistical tests, such as Greedy Forward Selection or Backward Elimination [20].

In the following paragraphs, we present an introduction to the fundamental characteristics of each regression model.

6.1.1 Linear Models

On linear regression models, it is expected that the unknown (dependent) variable can be expressed as a linear combination of many known (independent) variables. The generalized equation linear regression is:

$$\hat{y}(w, x) = w_0 + w_1 x_1 + ... + w_p x_p = \boldsymbol{w}^T \boldsymbol{x}$$

where $\hat{y}(w, x)$ is the predicted value, $\boldsymbol{w} = (w_0, w_1, ..., w_n)$ are computed model coefficients and $\boldsymbol{x} = (x_1, x_2, ..., x_n)$ are the independent variables. w_0 is also called intercept, and defines the predicted value if the input x vector is zero.

Standard linear regression models aim to find the coefficient vector $\boldsymbol{w}$ such that the residual sum of squares between the correct outputs from the dataset, and the values predicted by linear approximation, is minimized. In math terms, if $X \in R^{NxM}$ is a matrix, where N and M is the number of samples and features respectively, the function to be minimized is:

$$min_w ||X\boldsymbol{w} - \boldsymbol{y}||_2^2$$

where $\boldsymbol{y} = (y_1, y_2, ..., y_n)$ are the correct output values and $||...||_2$ is the Euclidean or L2 norm. This function is called **objective function** and is usually different for each machine learning algorithm.

One of the major drawbacks of the standard linear regression model is that it assumes that known variables are independent of each other. However, this is not true for `bzip2` workload, as it can be clearly seen from the hardware events signature presented in Figure 5.1 (at section 5.1. The same can be said for the hardware events behavior of other workloads, even though we do not actually visualize their behavior. This means that it is possible to predict the value of one variable from the values of multiple others used in the same model.

The phenomenon described above is known as **multicollinearity** and, when it is present, it leads to large variance of the predicted variable, as $\boldsymbol{w}$ values tend to have bigger values. One common and efficient way to address this problem is to impose a penalty on the size (or/and the number) of model coefficient values w.

Lasso

Least Absolute Shrinkage and Selection Operator (LASSO) is a regression analysis method that can entirely drop features, if this will lead to more accurate predictions [62]. This is achieved using a L_1 regularization term that imposes sparsity among the coefficients w of the model. The objective function of Lasso is the following:

$$min_w \frac{1}{2N}||Xw - y||_2^2 + \lambda||w||_1^2$$

The $\lambda \geq 0$ parameter controls the amount of shrinkage. As the value of λ gets larger, the model becomes more robust to collinearity. Prediction models produced using LASSO have generally fewer variables, which usually translates to lower variance and better model accuracy to unknown samples.

Elastic Net

Elastic net is a generalization of LASSO, as it combines the L_1 penalty from LASSO with an L_2 penalty[69]. In cases of high correlation among features, LASSO arbitrarily selects one and ignores the others. The addition of an L_2 regularization term in elastic net allows for a compromise between fewer and better selected variables in the end model [45]. The generalized objective function is:

$$min_w \frac{1}{2N}||Xw - y||_2^2 + \lambda_1||w||_1^2 + \lambda_2||w||_2^2$$

where both λ_1 and λ_2 control the coefficient size shrinkage, but only λ_1 imposes coefficient sparsity. One can easily see that LASSO is a special case of elastic net, for $\lambda_1 = \lambda$ and $\lambda_2 = 0$.

6.1.2 Tree-based Methods

Tree-based methods are non-parametric supervised learning methods that involve segmentation of the prediction space into a number of non-overlapping regions, which are formed using a set of decision rules inferred from the input data features. Usually, the tree methods work top-down by choosing, at each, step a variable / feature that "best" splits the set of items [48]. Therefore, the features perceived as contributing ones by the tree are chosen, while the rest are ignored. This property of in-built feature selection, is crucial for our datasets.

Being a non-parametric method, trees are also able to capture non-linear interactions between inputs and target values. Unfortunately, non-linear models are more vulnerable to **overfitting**, which is phenomenon that arises when the model learns the detail and the noise in the training set to the extent that it negatively impacts the accuracy of the model to unseen / new data [5].

Even though tree-based methods are usually preferred for classification problems (i.e. distinct number of output values), it is possible to support regression problems by choosing an appropriate splitting metric. Both classification and regression trees, however, use the same concept of **node impurity**, which is a measure of homogeneity of the target values at a specific tree node.

Decision Tree (CART algorithm)

Classification and Regression Tree (CART) algorithm builds a binary tree, where each root node best splits the tree node with the goal of maximizing the information gain among all possible splits [4]. A small set of rules is present in each tree node, which determines the next node (i.e. child node) on the path to the leaf nodes. The leaf nodes of the CART tree contain a single output value $\hat{y}$ which is the used for the prediction.

CART algorithm uses the Variance Reduction metric to evaluate the quality of split at each step (i.e. impurity function), when the target variable is continuous:

$$I_V = \frac{1}{N}\sum_{i=1}^{N}(y_i - \bar{y})^2 \quad \text{where} \quad \bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i$$

The information gain is defined as the difference between the parent node impurity and the weighted sum of the two child node impurities [48]. Mathematically, this translates to:

$$IG(\boldsymbol{D}, \boldsymbol{D}_{left}, \boldsymbol{D}_{right}) = I_V(\boldsymbol{D}) - \frac{N_{left}}{N} I_V(\boldsymbol{D}_{left}) - \frac{N_{right}}{N} I_V(\boldsymbol{D}_{right})$$

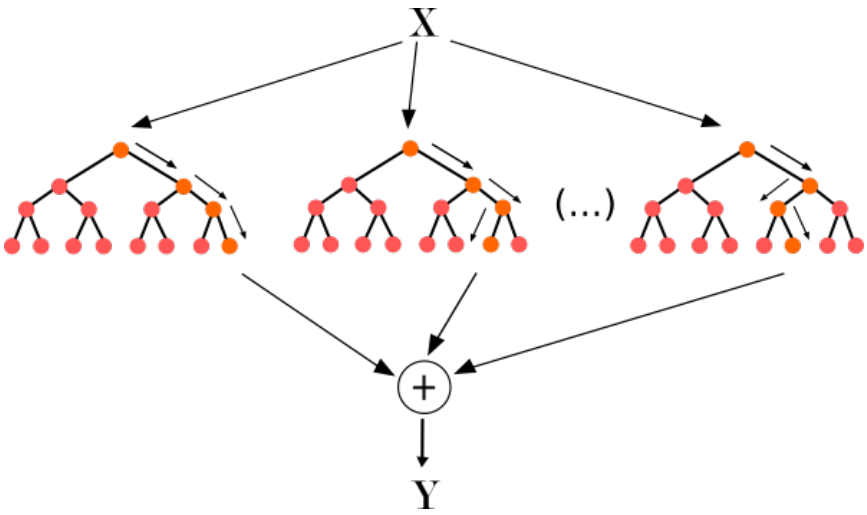


FIGURE 6.1: Random Forest ML algorithm visualization

where $\boldsymbol{D}$ is the dataset with N samples in the parent node, and $\boldsymbol{D}_{left}$ and $\boldsymbol{D}_{right}$ are the datasets resulting from partitioning $\boldsymbol{D}$, which have N_{left} and N_{right} samples respectively. If the information gain value is maximized in each tree node, then the decision tree makes better predictions. Finally, the above procedure stops when the tree has reached its maximum width, which is a user-defined parameter given before the training phase.

Random Forests

Random forests are one example of ensemble learning methods, which typically combine multiple individual predictions from a number of base models to compute the final prediction [27]. In this case, the base models are decision trees which have been trained independently using a randomly selected subset of all features (with the same CART algorithm). This allows for the construction of a diverse set of regression trees that overcomes the overfit problem found in simple decision trees [21].

Figure 6.1 illustrates the structure of a random forest model. The input matrix $X \in R^{NxM}$ consists of N samples with M features each, and it is fed to each base model (i.e. decision tree) independently. Note that the path from the root to the leaf varies for each tree, as the node conditions are different. Eventually for every tree the path arrives on a single leaf node. The final predicted value is just the average of all individual leaf node values.

6.1.3 Artificial Neural Networks

Artificial Neural Networks (ANNs), or more simply neural networks, are computational systems of interconnected "neurons" that "learn" to perform tasks without being explicitly programmed by a set of specific rules. Instead, they extract their

knowledge by detecting patterns and associations in data, through a training phase. In addition, they automatically generate new features by combining existing ones to make more accurate predictions. This is achieved by simulating the learning behavior of biological neural systems (e.g. animals' brain), which they draw inspiration from [23].

The artificial "neurons" that constitute an ANN, which have the role of **processing elements** (PE), are connected with each other to collectively process the information. These connections are also called **edges** or **arcs**. This is where the power of ANNs comes from. In a typical neural network there can be hundreds of such computational units, which can be connected in various ways. The different organization and connection of processing elements leads to diverse network architectures, which eventually might be more suitable for more specific tasks (e.g. image recognition, speech recognition, etc.)

Figure 6.2 visualizes the basic components of each processing element. At first, each PE is fed with weighted inputs (i.e. coefficients) from the units that connect to it. Then these input weights are summed and fed into the **activation function** and a single output value is produced, which is then passed to the next units. The choice of the activation function, which introduces non-linearity to the network, can greatly affect the behavior and the accuracy of the model.

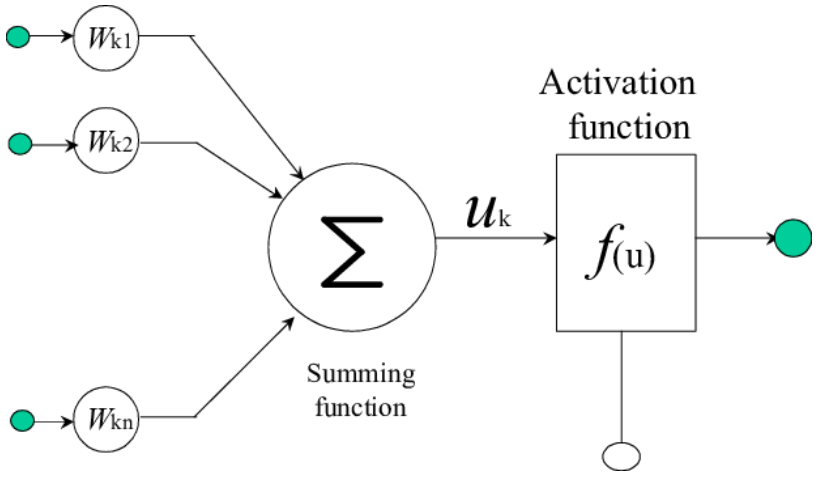


FIGURE 6.2: Components of a neural network processing element

An ANN is organized in layers that contain many processing elements. Traditionally a connection between two processing units exists if these belong to different layers. However, more modern neural network architectures allow for self-feedback connections. At the very least, each ANN is composed by the following layers:

- **Input layer** – The features of each sample are fed directly into the input layer. The number of processing elements in this layer is usually equal to the number of features, as each unit receives a single value (i.e feature). By being the first layer of the ANN, there are no actual computations performed in these nodes

as the only purpose of input layer is to pass the sample values to the next (hidden) layers.

- **Hidden layer(s)** – In these layers the actual "learning" process takes place, as the values are transformed. Depending on the ANN architecture, there can be a single or multiple hidden layers. In any case, the input layer values are fed into the processing units of the first hidden layer. Then, as described previously, the weighted sum is computed and it is passed over the activation function to produce a single value. The resulting value is propagated to either the next hidden layer, if there is one, or the output layer.
- **Output layer** – This is the last layer of the neural network and where the final prediction value is formulated. This layer may contain one unit, in cases of binary classification and regression, or multiple, in case of multi-label classification. Furthermore, for regression problems the transformed value of this unique output unit is not passed through the activation function.

During training, edge coefficients are optimized to minimize the prediction error. For that purpose, ANNs use an **optimization function**, which aims at finding the optimal edge coefficient values with as few iterations as possible. A very common choice is the **stochastic gradient descent** [49], or any of its modern variants [17, 36]. In order for the optimization function to properly adjust the coefficients, it is crucial to know how good are the predictions of the network. For that purpose, a **loss function** is employed, which quantifies the cost of a misprediction. The larger the loss function value, the larger the error between the correct value and the predicted one.

More specifically, the following iterative procedure is followed:

- At first, a **forward pass** of the network is made, using a single sample to make the prediction. This means that an input vector is fed to the neural network and propagates through the PEs, layer by layer, until eventually reaching the output layer. There, the final prediction value of the network is computed. Then, the error between the network prediction and the correct value is calculated, using the provided loss function.
- Once the prediction error is known, a **backwards pass** of the network is performed, starting from the output layer. Then, the error of each PE is computed, which reflects its contribution to the final prediction value. These values are then used by the backpropagation algorithm [52] to derive the gradient of the loss function. Finally, the gradient values are fed into the optimization algorithm, which adjusts the weight coefficients. Hopefully, the new coefficients will give better predictions.

Latest advances in hardware made it possible to feed multiple samples simultaneously (i.e. **mini-batches**) to the network during the forward pass phase. By using

multiple samples, a potentially single noisy sample will not harm the weight adjustment, as new coefficients will be computed based on the average prediction error of the whole batch. This technique is called **batch normalization** [34] and uses a modified version of gradient descent, which leads to slower but more steady convergence.

Multi-Layer Perceptron

A **multi-layer perceptron** (MLP) is a class of feedforward neural network, as there are no backwards connections between the processing elements [21]. An MLP is one of the simplest structures of ANNs that are still used today. MLPs typically consist of fully-connected layers and can have one or more hidden layers. A fully-connected network means that each processing unit that belongs to a layer connects to all units of the next layer. When an MLP consists of two or more hidden layers and the processing elements use non-linear activation functions, then the network can be proven to be a universal function approximator [11].

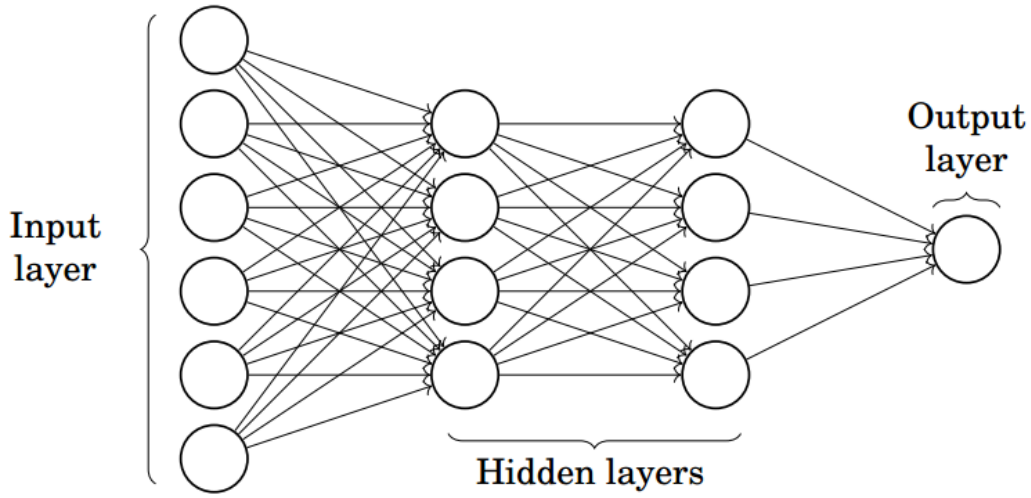


FIGURE 6.3: Visualization of a multi-layer perceptron with two hidden layers

Figure 6.3 shows an MLP network with two hidden layers. Note that in contrast to what is shown in the figure, the number of processing elements on each hidden layer can be different.

MLPs can be used for both classification and regression. In classification problems, each processing element uses a non-linear activation function. In regression problems, however, where a continuous value shall be predicted, the output layer processing elements do not use an activation function.

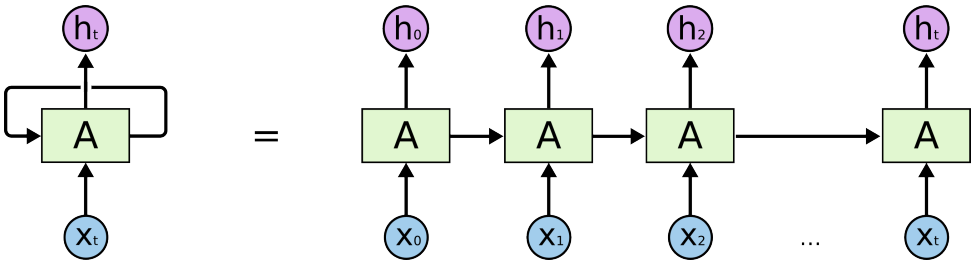


FIGURE 6.4: Unrolled recurrent neural network

Long Short-Term Memory (LSTM) Networks

Before diving into Long Short Term Memory (LSTM) networks, we discuss Recurrent Neural Networks (RNN). Recurrent neural network is a class of ANNs which is able to model dynamic temporal behavior. In contrast to feedfordward neural networks, RNNs can use their internal memory to process arbitrary sequences of inputs. An RNN uses a chain-like structure, where the input of a processing unit depends on the value of the previous input value. Figure 6.4 shows the organization of the processing elements of an RNN. They can be thought of multiple copies of the same network, where each one passes a "message" to the next network.

An LSTM network is a special type of RNN, which was introduced to mitigate one major RNN drawback [28]. Due to their design, RNNs experience poor performance when there are long-term dependencies on the data. This is partly caused by the vanishing gradient problem [46], which makes it difficult to effectively update the coefficients of the first layers as the network becomes deeper. LSTMs have been carefully architectured to avoid this problem, as they consist of memory units that can store more information compared to RNNs. More specifically, each LSTM cell consists of the following gates:

- **Forget gate**, which decides what information from the previous cell is worth remembering and forgets what is irrelevant.
- **Input gate**, which selectively updates the cell states depending on the values of the input data.
- **Output gate**, which decides which part of the cell state is going to be propagated to the next cell.

LSTMs have been successfully used in a wide variety of problems that utilize time-series data, since these cells remember important events over arbitrary large time intervals. In the scope of this thesis, we are going to evaluate their performance to predict the value of the latest sample of CPU core voltage using multiple hardware event measurements from previous samples, too.

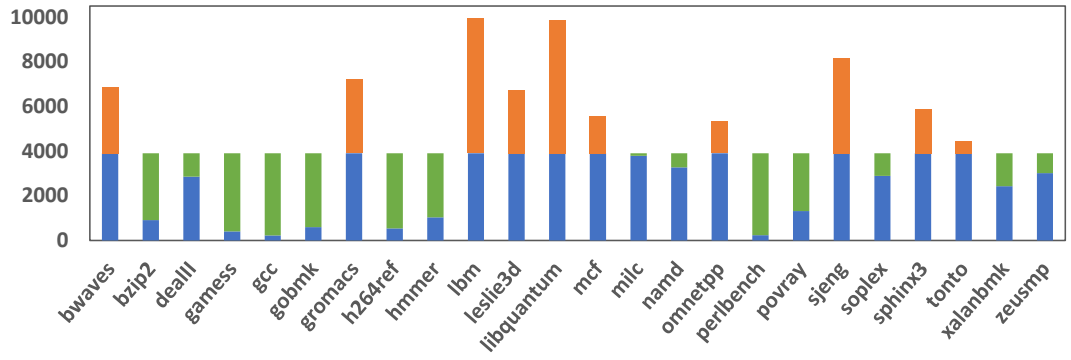


FIGURE 6.5: Number of duplicated/dropped samples per workload. The green bars represent the number of duplicated samples, and the orange bars the number of dropped samples

6.2 Data Pre-processing

Before feeding (training and testing) data as input to each model, it is vital to perform the following pre-processing steps to acquire better estimation accuracy.

6.2.1 Data Balancing

Both **HLM** and **AM** datasets consist of samples from many workloads. The execution time of each workload, however, is different and ranges from 23 seconds to 10 minutes. Therefore, the number of samples taken for each workload varies. In order to not be biased towards the workloads that run for longer time periods, we ought to balance the dataset samples. Different approaches have been introduced to tackle this problem, such as generating synthetic samples [9], or penalizing machine learning models to make them pay more attention to minority classes [2].

Unfortunately, these approaches cannot be effectively used in our case, since they primarily address classification problems (not regression ones). Hence, we decided to resample the dataset to balance workload classes. This is achieved by under-sampling the workloads with more samples and, at the same time, over-sampling those with less. More specifically:

- We calculate the **average** number of samples over all workloads.
- For the workloads that have more samples than the mean, we randomly **drop samples** (i.e under-sampling) until this number is equal to the mean.
- For the workloads that have less samples than the mean, we randomly **duplicate samples** (i.e over-sampling) until this number is equal to the mean.

By following this process, we end up with equal sized profiles for each workload. Figure 6.5 shows how many samples are dropped (orange bar) or duplicated (green bar) for each workload.

6.2.2 Data Normalization

Data normalization is a very common and often necessary data pre-processing step. Many machine learning algorithms behave significantly better if all input features are scaled to the same range, as comparisons between different features are fair. Two methods are well known for scaling data:

- **Normalization**, which scales all feature values to range $[0, 1]$
- **Standardization**, which transforms all feature values to have **zero mean** and **unit variance**

In our experiments, standardization proved to give better results. Given a feature value x, then the standardized x_{new} value is:

$$x_{new} = \frac{x - \mu}{\sigma}$$

where μ and σ are the mean and standard deviation of feature values, respectively. It is worth noting that in order to avoid adding bias to the models, we utilize only the training data to calculate μ and σ values.

6.3 Model Parameters

In this section we summarize the parameters used for training the models. For the linear and the tree-based models, we used the following parameters:

- **Lasso**: λ_1 is set to 1.0
- **Elastic Net**: λ_1 is set to 1.0 and λ_2 to 0.5
- **Decision Tree**: The decision tree height is set to 4, as larger values lead to overfitting for our dataset. The number of decision rules per branch is set to $\sqrt{M}$, where M is the number of features.
- **Random Forest**: Each tree has the same parameters as a single decision tree. The final value is computed by taking the average value of 8 separate trees, which were trained on random subset of our training data subset.

6.3.1 Neural Networks

Activation Function

As described previously, the activation function defines the output of a single processing unit. Therefore, its choice can greatly affect the behavior of the neural network. Many different functions have been proposed over the years (e.g. sigmoid,

tanh), yet the most popular of all is the Rectified Linear Unit (ReLU) [24]. ReLU was introduced to effectively tackle the vanishing gradients problem, which was observed when sigmoid or tanh were used. Mathematically, it is defined as:

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

Using a ReLU activation function also has computational benefits, as the computation of exponentials on sigmnoid or tanh is expensive. This resulted in faster network training times.

Optimization method

We choose the Adam optimization algorithm [36], which is a widely-used modern variant of the classical stochastic gradient descent method. Instead of using a single learning rate (α parameter) for all features, as stochastic gradient descent does, it maintains a single parameter for each network weight (coefficient) and separately adjusts their values. This modification greatly improves performance and makes it ideal candidate for problems with many features. For our evaluation we use the default parameters, presented in the original paper, which are $\alpha = 1e-3$, $\beta_1 = 0.9$, $\beta_2 = 0.999$.

Regularization

Regularization is a technique that is used to control the capacity of neural networks to prevent overfitting. Similar to the Lasso and Elastic Net models, a regularization term is added to **network weights**. In the case of an L_2 term, the magnitude of all parameters is penalized. On the contrary, the use of an L_1 term has a property that it leads the weight vectors to become sparse during optimization (i.e. use a subset of input features). It is also possible to utilize both regularization terms on the same network. Especially for neural networks, L_2 regularization is expected to give superior performance over L_1.

Nonetheless, we experimented with L_2 values of 0.01, 0.05, 0.1 and various L_1 values ranging from 0.01 to 1. In the majority of our tests, however, using just L_2 regularization gave better results than the combined L_1, L_2 regularization.

The architecture of both MLP and LSTM networks consist of **two hidden layers** with **128 processing units** each. For the MLP network the inputs belong to the same interval as the core voltage value. In contrast, the LSTM is fed with hardware events from the **last 4 intervals** in order to explore if previous interval info could result in better accuracy.

6.4 Model Evaluation

6.4.1 Cross-validation

Cross-validation, or out-of-sample testing, is any statistical method that can be used to evaluate the performance of a machine learning model on unseen data. It is frequently used for prediction tasks to get an insight of how the trained model will generalize in practice on an independent dataset. Typically, one iteration of cross-validation consists of reserving a part of the dataset, training the model with the rest of the dataset, and finally, evaluating its accuracy using the reserved part. Common problems in case of bad model performance are overfitting and selection bias [6].

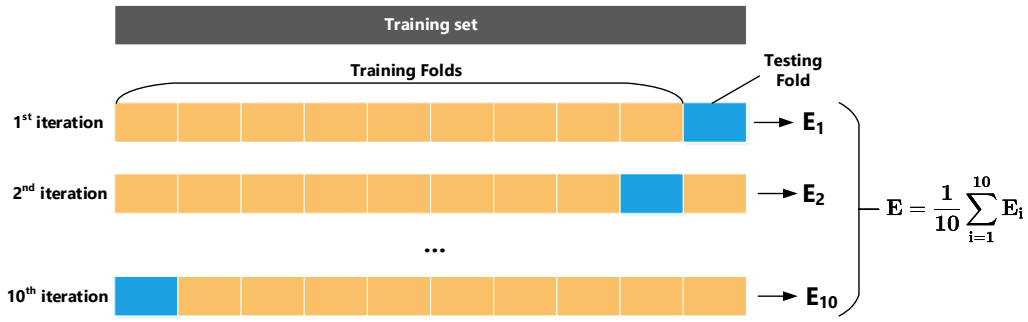


FIGURE 6.6: 10-Fold Cross-Validation Procedure Visualization

To evaluate our models we utilized the **k-fold cross-validation** procedure, which consists of the following steps:

1. Shuffle the samples of the entire training set randomly
2. Split the shuffled training set into **k folds** (or groups)
3. For each such group:
   - Pull this group **out** of the training set
   - **Train a new instance** of the model with the rest of the training set
   - Test model performance **using the group that was taken out**
   - **Evaluate** model performance by computing its prediction error
4. Calculate the overall cross-validation error

Figure 6.6 visualizes the whole procedure and shows the formula for computing the final cross-validation error. This error term can be effectively utilized as a robust performance metric of the trained model. In our experiments we set $\mathbf{k = 10}$, which is commonly used, as its produced error term does not suffer either from excessively high bias, or from very high variance [35].

6.4.2 Loss Function

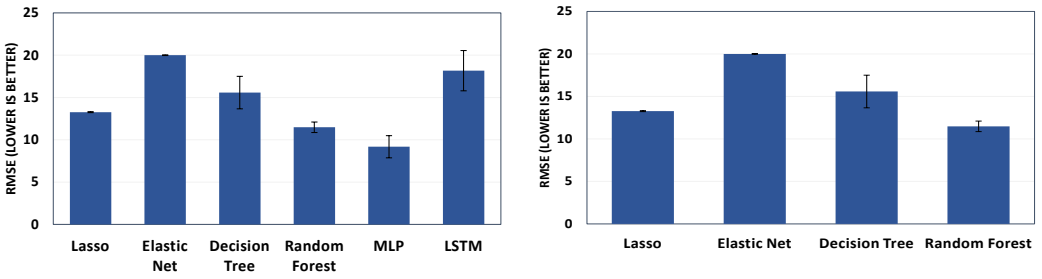
Mean Square Error (MSE) is the single most popular loss function for measuring accuracy on regression problems. It measures the average magnitude of the errors in a set of predictions, as it computes the average of squared differences between prediction and actual observation:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

where N is the number of samples, y_i and $\hat{y}_i$ are the actual and predicted value, respectively. Taking the average squared errors has some interesting implications for MSE. Since the errors are squared before they are averaged, the MSE gives a relatively high weight to large errors, which is desirable in our case.

6.5 Core Voltage Estimation Results

Instead of using MSE to quantify how good our models estimate the core voltage value, we will use the Root Mean Square Error (RMSE), which has the same units as the quantity being estimated. Figure 6.7 presents the estimation error for each ML model on both **HLM** and **AM** datasets. Since we are evaluating performance through error, lower is better in this case. The baseline error, which originates from always predicting the same average core voltage value, is $\sim$ 25 millivolts on both datasets.



(A) **HLM** dataset (B) **AM** dataset

FIGURE 6.7: Voltage estimation accuracy of ML models

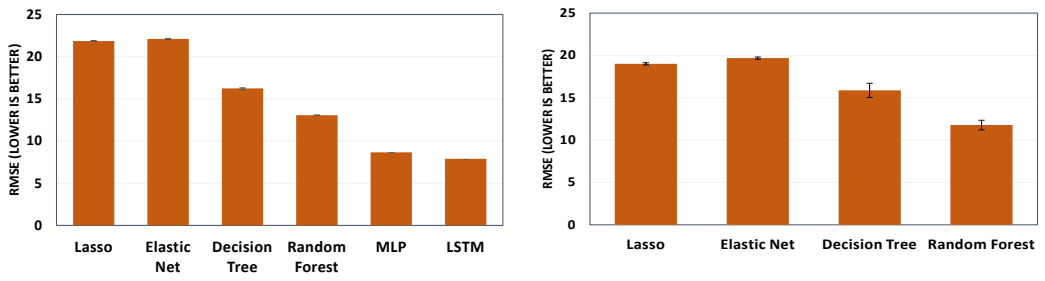
Note that for the case of the **AM** dataset no error bars are shown for the neural network models. Despite our best efforts and experimentation with a wide variety of L_1 and L_2 regularization terms, we were unable to acquire acceptable prediction accuracy with any of the tested parameter values.

Now, focusing on the plots themselves, we make the following observations:

- LASSO, Random forest and MLP **perform the best**, as the estimation error is lower than that of their counterparts. From these three, MLP achieves the best accuracy, while the accuracy of LASSO and Random Forest is very close. It is worth noting, however, that the training phase of MLP took considerably longer time (i.e. $\sim$ 20x) than the training phase of the other two.
- The performance of Elastic Net is really bad, considering that it is a generalization of LASSO. The same can be observed about the LSTM network performance, which also experiences large variance in its predictions.
- The performance of the models trained on **AM** dataset is on par with the performance of the ones trained on the **HLM** dataset. Therefore, monitoring a large number of very specific hardware events does not lead to better predictions, and thus the high-level metrics should be preferred.

When reviewing which hardware events were selected by our ML models, we observed that power events were dominating. On both linear and tree-based models, these events were preferred over all others, and thus, they were given very large coefficient values. This should come as no surprise, as there is a direct physical relationship between power consumption and core voltage value.

Nonetheless, we wanted to explore if it would be possible to estimate the core voltage value, using solely performance events, and how these models would perform. Figure 6.8 shows the accuracy of our ML models, when power-related features are not used as inputs. Since the dataset output values do not change when we drop certain features, the baseline is the same as previously.

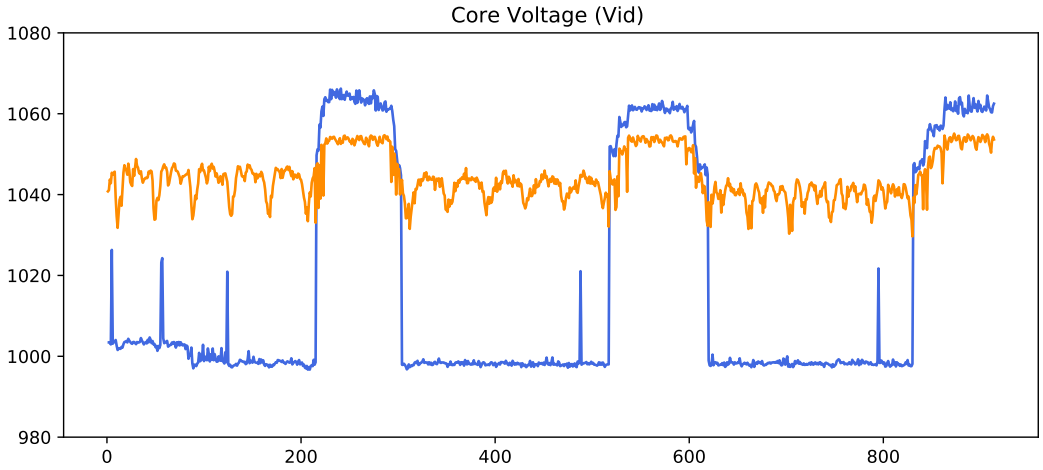


(A) **HLM** dataset (B) **AM** dataset

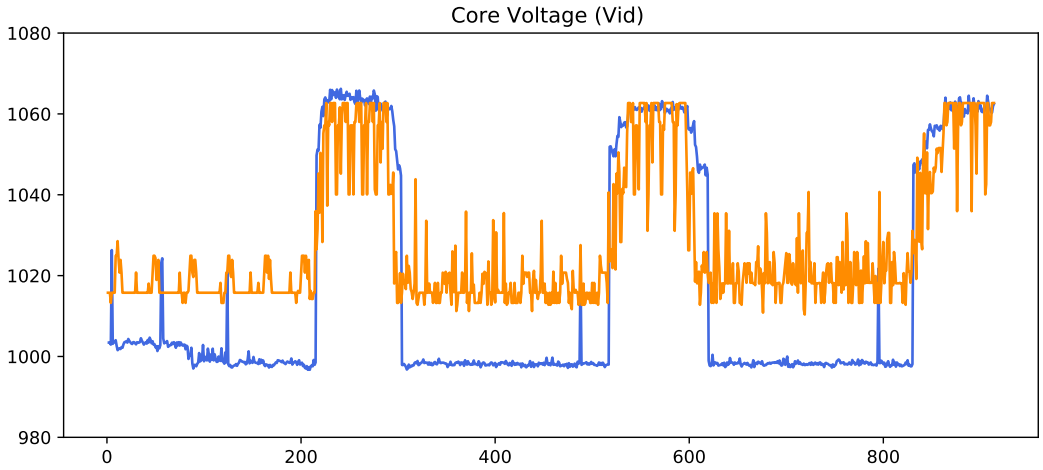
FIGURE 6.8: Voltage estimation accuracy of ML models (no power events)

We make the following observations:

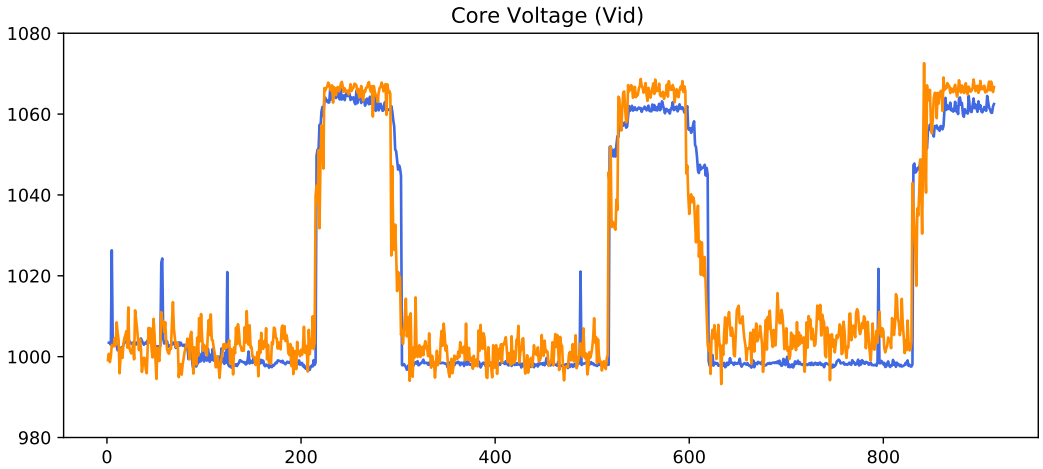
- Their is a clear distinction among the different ML models, as the neural networks perform the best, tree-based are in the middle, while linear models perform the worst. In this case, Elastic Net accuracy is on par with the LASSO model.



(A) LASSO



(B) Random Forest



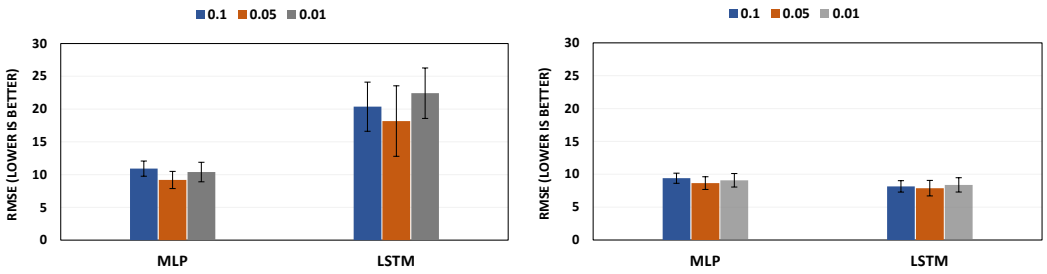
(C) Multi-Layer Perceptron

FIGURE 6.9: Core voltage estimation of LASSO, Random Forest and MLP models on bzip2 workload

- In contrast to the previous evaluation where we utilized the power events, LSTM networks seem to have slightly better prediction accuracy. However, they still require about $5x$ more time to be trained and take about $2x$ longer to make the predictions than MLP, which makes them unappealing. MLP has the same prediction error, yet more stable behavior across multiple folds.
- The linear models trained on the **AM** dataset perform somewhat better those trained on **HLM** one. Surprisingly, the opposite true for the tree-based model. However, again we argue that it is not worth monitoring all hardware events, as these cannot be effectively used to derive better prediction models.

Figure 6.9 visually compares the LASSO, Random Forest and MLP model predictions, when they are trained without power events on bzip2 workload. The blue line represents the actual (i.e. measured) core voltage value, while the orange line the estimated one. The x-axis represents the samples taken every 100ms, while the y-axis shows the core voltage value. LASSO model completely misses the voltage fluctuations, while the Random Forest does a better job at capturing this behavior, however at the cost of increased noise. MLP manages to closely follow the voltage signature on the entire execution, with the addition of minor ripples during periods of stable voltage behavior.

Finally, since our experiments show that MLP and LSTM are undoubtedly the best prediction models, we decided to try to optimize their performance by fine-tuning the L_2 regularization term during the training phase. Figure 6.10 shows the estimation error of these models for various L_2 values (i.e. $0.1, 0.05, 0.01$), when the **HLM** dataset was given as input.

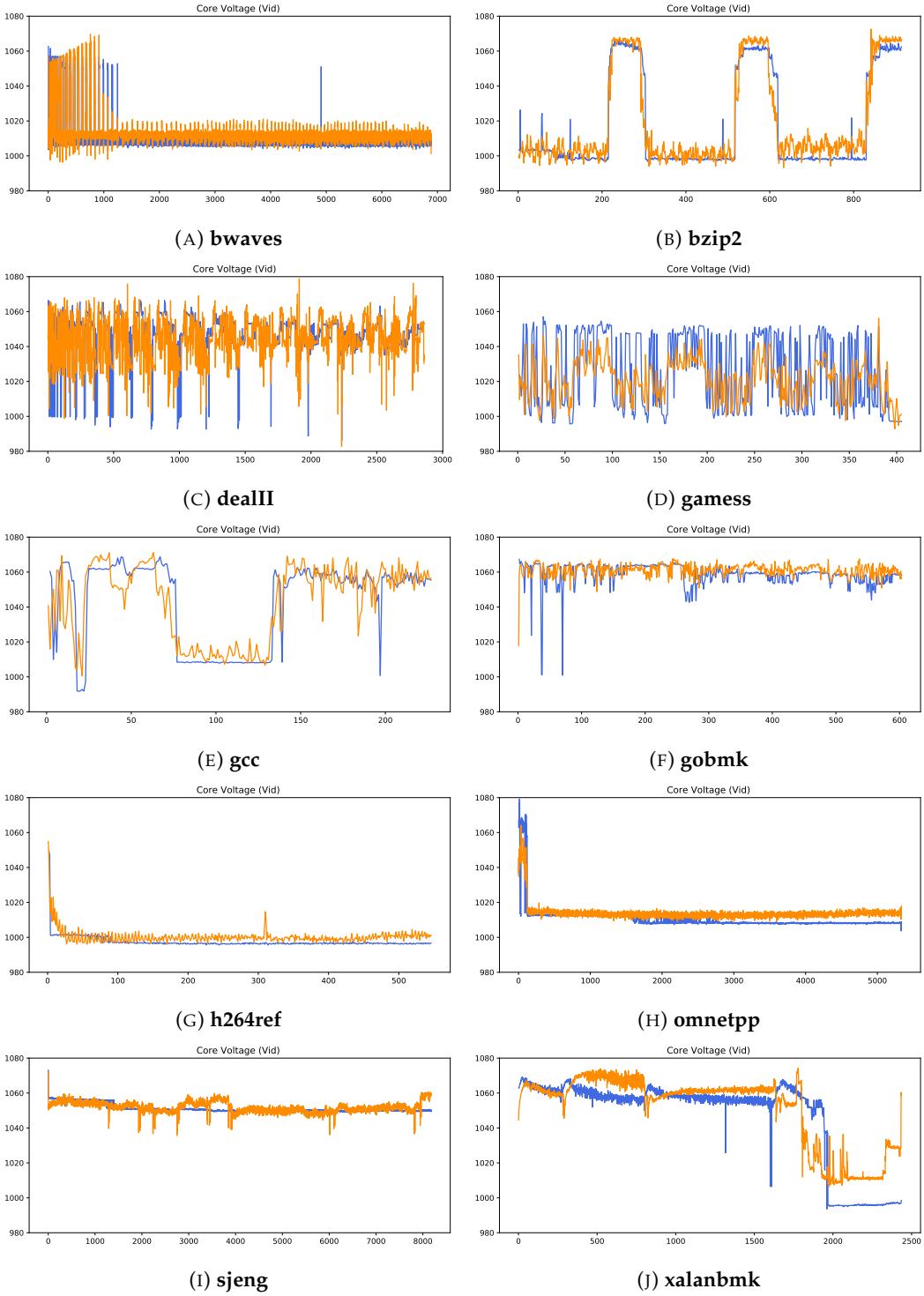


(A) **HLM** dataset (B) **HLM** dataset

FIGURE 6.10: MLP estimation error with various L_2 regularization term values for **HLM** dataset

For every case, training the models with $L_2 = 0.05$ leads to better accuracy. Figure 6.11 shows the quality of estimations of an MLP model, trained with the above parameter value, on numerous workloads. Again, the x-axis represents the samples taken every 100ms, while the y-axis is the core voltage value. For most workloads (i.e. bwaves, bzip2, gcc, gobmk, h264ref, omnetpp, sjeng) the model captures core voltage behavior with acceptable accuracy. On dealII and gamess workloads, the

very rapid peaks cannot be captured efficiently, yet the predictions remain within the range of these fluctuations. Unfortunately, the model does not give good predictions for the xalanbmk workload, despite its relative stable behavior. This might be due to the simultaneous change of the all hardware events in the same direction (i.e. correlation among all hardware events), which in turn negatively impacts the model predictions.



(A) **bwaves** (B) **bzip2**

(C) **dealII** (D) **gamess**

(E) **gcc** (F) **gobmk**

(G) **h264ref** (H) **omnetpp**

(I) **sjeng** (J) **xalanbmk**

FIGURE 6.11: Voltage estimations made by MLP model trained with $L_2 = 0.05$, on a subset of SPEC CPU2006. X-axis represents samples taken every 100ms, while y-axis is the core voltage value

Chapter 7

Online Program Phase Detection

At run-time, programs exhibit varying behavior as they may alternate between periods of different execution characteristics. Depending on the application, these may last from few milli-seconds to many seconds or even minutes. Periods where application behavior is relative stable, with regard to some metric, either program-specific or quantifying the interaction with hardware, can be defined as program phases. Note that we are not referring to actual code segments, but rather to temporal microarchitectural phases, which are formed due to the application interaction with the hardware. Researchers have been trying to capture, understand and characterize these phases, in an attempt to further unleash opportunities for software and hardware optimizations [15].

Online phase detection methods have been developed, in order to enable adaptive system optimizations. These methods are applied at application run-time, and by utilizing profiling techniques they are able to detect phase changes very fast. Examples of dynamic phase-guided optimizations include smart cache resizing [1], more efficient thread scheduling on heterogeneous multi-processor systems [60], improved memory access times by predicting data locality [55] and faster program simulations [56]. Note that phase detection methods do not literally detect application phase changes. Instead they detect variations in application behavior which are assumed to be the outcome of a phase change [15].

In this chapter, we introduce a novel online phase detection mechanism, which exploits automatic, hardware-guided core voltage and frequency adjustments of Intel `SpeedShift` DVFS technology. By monitoring the core voltage and productive frequency values, we are able to effectively detect phase changes with very low runtime overhead. We evaluate our tool phase detection accuracy and we perform a comparison with a state-of-the-art tool (ScarPhase [54]).

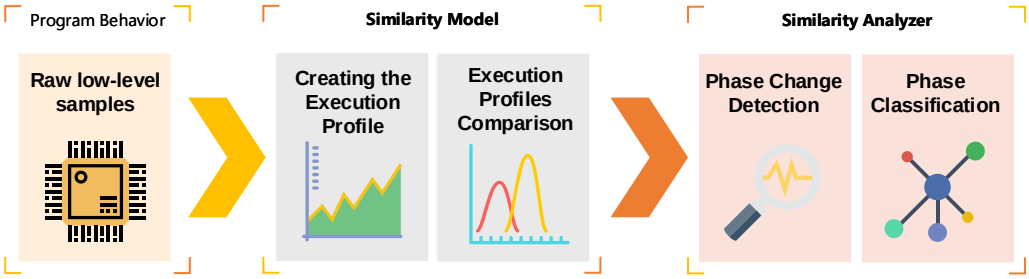


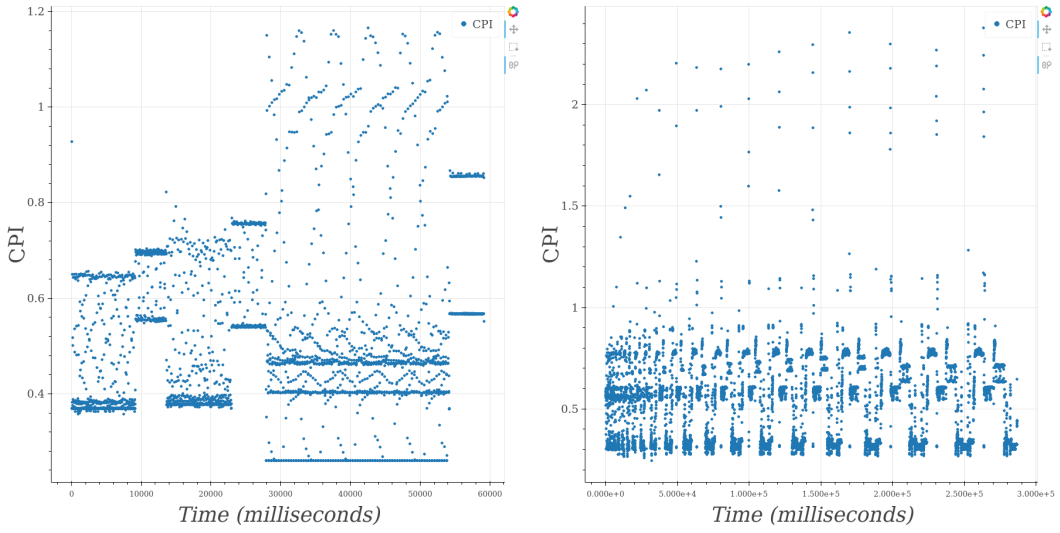
FIGURE 7.1: Framework for detecting program phases

7.1 Framework

The majority of the proposed online program phase detection methods are based on the same abstract framework. Figure 7.1 illustrates the general workflow, which typically consists of the following steps:

- At first, the method monitors application execution using low-level metrics, for a certain time period or until a certain count of occurences of some hardware event. Usually methods collect many samples during an interval, in order to better capture the dynamic characteristics of the application.
- The raw values are then transformed into a more meaningful representation (e.g. matrix, vector, histogram), which encapsulates the knowledge gathered from all samples during this interval. This is also known as the **execution profile**. This allows for easier comparison of pplication behavior among different intervals.
- The execution profile of the latest interval is compared against other execution profiles. Depending on the actual method, these may be only the most recent ones or may originate from the entire past execution. This process produces a **similarity value** for each such pair, which represents the similarity between the two execution profiles. The larger this value, the more similar the two profiles are.
- The respective similarity values are then passed to a **similarity analyzer**. At the very least, the similarity analyzer checks these values against a static threshold and decides whether a phase change has occurred or not. Again, depending on the method, the analyzer may dynamically change the value of the threshold, based on statistics gathered from previous intervals.

The above process is repeated until the application terminates. Note that we omitted the last step of the similarity analyzer, which is the **phase classification**. This step is optional but is commonly found on online phase detection methods. More specifically, it is possible for the similarity analyzer to keep track of past execution profiles along with phase identifiers. Doing so, it can also determine on which phase the last



(A) **bzip2** (B) **dealII**

FIGURE 7.2: CPI variability during execution of bzip2 and dealII workloads

profile belongs to, even if the profiles of that phase have been sampled way back in the past. This extra logic leads to phase classification methods, which are really practical for programs that contain small and repetitive phases.

The majority of online phase detection and classification methods aim at detecting non-overlapping phases which consist of fixed-size intervals. However, instead of using time to define the size of intervals (e.g. each interval lasts for 10 or 100 milliseconds), the research community often opts for using the number of executed instructions. Early studies used intervals of 100 million instructions, which is still used today in order to simplify the comparison of different methods.

Earlier we defined program phases as periods of relative stable behavior with regard to some metric. Most researchers agree that Cycles per Instructions (CPI) can be effectively used to describe the "real" program phases (i.e. ground truth) [15], especially when these phases will be exploited for reducing the time of simulating application behavior. However, CPI should not be considered as the universal metric that "defines" program phases, as this greatly depends on the relevant optimization itself. Figure 7.2 shows how CPI changes over time for `bzip2` and `dealII`. Note that even though CPI values can vary greatly, there are intervals of relative stability. Although CPI might be a good metric for sequential applications, it should never be used for detecting phases in multithreaded applications, as shown in [47], due to its large variability on loaded SMT systems.

7.2 Execution Frequency Vectors

Numerous studies have shown that there is a strong correlation between the code executed and the behavior of the application. Hence many program classification techniques have employed **execution frequency vectors** [58, 56, 15]. These vectors typically capture which code segments are being executed during an interval. Earlier studies [15] used individual instructions, which were hashed into a large bit vector (i.e. Extended Instruction Pointer Vector – EIPV). By computing the Manhattan Distance:

$$D(V_1, V_2) = \sum_{i=1}^{N} |V_1(i) - V_2(i)|$$

between execution frequency vectors V_1 and V_2, one could find how similar (or rather dissimilar) were these execution profiles. Unfortunately though, as these vectors get large, the storage requirements as well as comparison overheads are becoming limiting.

In contrast to individual instructions, researchers in [56, 58] utilized **Basic Block Vectors** (BBV), which capture code execution context at a coarser granularity than plain instructions. Basic blocks are continuous code segments with no branches in, except to the entry, and no branches out except at the exit [25]. Moreover, they showed that hashing basic block addresses into a vector of just 32 counters is enough for practical purposes. This can lead to reduced overheads due to the small size of the vector. The similarity values are again obtained using the Manhattan Distance between the BBVs.

Still, however, none of execution profiles is suitable for online phase classification, as each individual executed instruction or each basic block, needs to be sampled. One straightforward way to avoid this, is to collect sparse samples as discussed in [12]. In this study, an estimate of EIPV is computed, as instructions are collected using stratified sampling. A more recent study [54] takes samples of branch addresses, with the goal of approximating BBVs values. By leveraging modern performance monitoring capabilities, such as Precise Event Based Sampling (PEBS), they demonstrate that online phase classification is achievable with low runtime overhead and good accuracy. Finally, they have open-sourced their tool `ScarPhase`, which we will use as baseline in our evaluation.

In the following paragraphs, we describe our approach, pointing out differences between ours and previous work when needed.

Metric	# of Entries	Lowest value	Highest Value	Bin Size
Core Voltage	32	962 mV	1080 mV	4 mV
Productive Frequency	64	0 MHz	3000 MHz	47 MHz

TABLE 7.1: Core voltage and productive frequency histogram parameters

7.3 Our Approach

Instead on relying on the code section of the program that is being executed, we base our approach on hardware core voltage and productive frequency measurements, and we argue that they can be used, in most cases, to efficiently detect phase changes.

7.3.1 Sampling

Since the core voltage and productive frequency are updated every $\sim$ 1 millisecond, we choose this value as our sampling frequency. For each interval of 100 million instructions, we take multiple samples that are stored in the memory. Note the number of samples taken depends on the time period that it takes for the processor to execute 100M instructions. Therefore, the number can significantly vary among different workloads as well as between different code sections of the same workload. For the majority of the workloads in our evaluation each interval consists from 10 to 20 samples, while there are cases of individual intervals with over 100 samples.

7.3.2 Execution Profile

Once the interval ends, these samples are combined to create two histograms, one for each metric. The core voltage at the fixed 3.0GHz maximum stock frequency ranges from $\sim$ 970 to $\sim$ 1080 millivolts, while the productive frequency ranges from 0 to 3000 MHz. For every sample we find the correct histogram bin, which are of equal widths (or sizes). The number of histograms bins for the core voltage distribution is set to 32, which is big enough to capture even small (i.e. up to 4mV) fluctuations. However, due to the larger range of productive frequency values, we use a larger histogram of 64 bins, which achieves a more accurate representation. In our experiements, more bins did not led to negligible improvements. Table 7.1 summarizes the parameters for both histograms. Finally, as the number of samples per interval varies, histograms are normalized (i.e. divided by the number of samples).

7.3.3 Histogram Similarity

The next step is to calculate the similarity between two histograms, in order to decide weather a new phase has began. One idea is to reuse the Manhattan Distance that

was previously used in the BBVs, or some other popular distance metric (e.g. Euclidean Distance). However, these metrics are agnostic of the overlapping between two histograms. Therefore, we use the algorithm presented in [8], which computes a distance between sets of measured values. This **distance acts as a measure of dissimilarity** between the histograms. Its major advantage is that it takes into account the similarity of both non-overlapping and overlapping parts of the distributions, in contrast to traditional metrics.

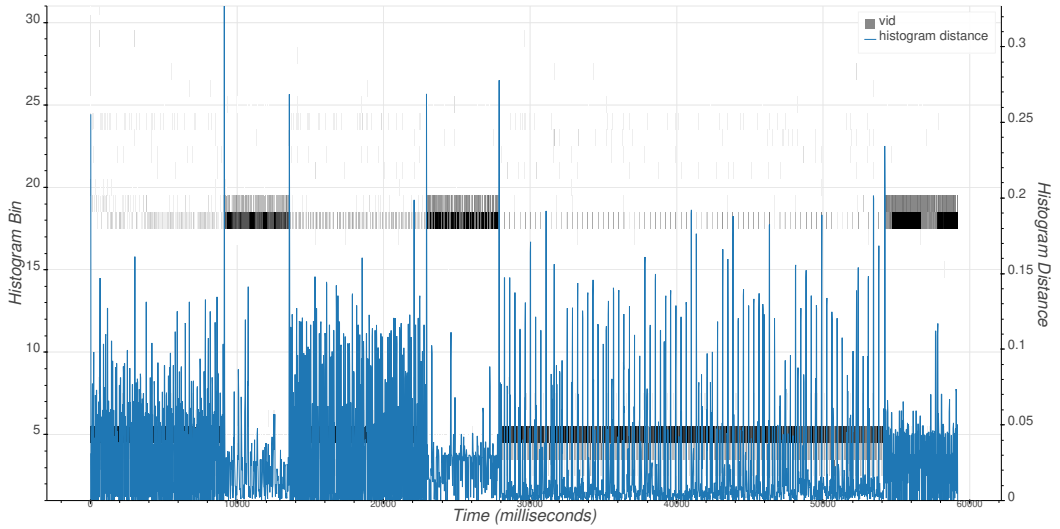
input : Two normalized histograms H_1, H_2 of size N
output: The normalized histogram distance
1: $prefixSum = 0$
2: $histDist = 0$
3: **for** $i = 0$ **to** $N - 1$ **do**
4: $\quad prefixSum \mathrel{+}= (H_1(i) - H_2(i))$
5: $\quad histDist \mathrel{+}= |prefixSum|$
6: **end for**
7: **return** $histDist/(N-1)$

Algorithm 1: Histogram Distance

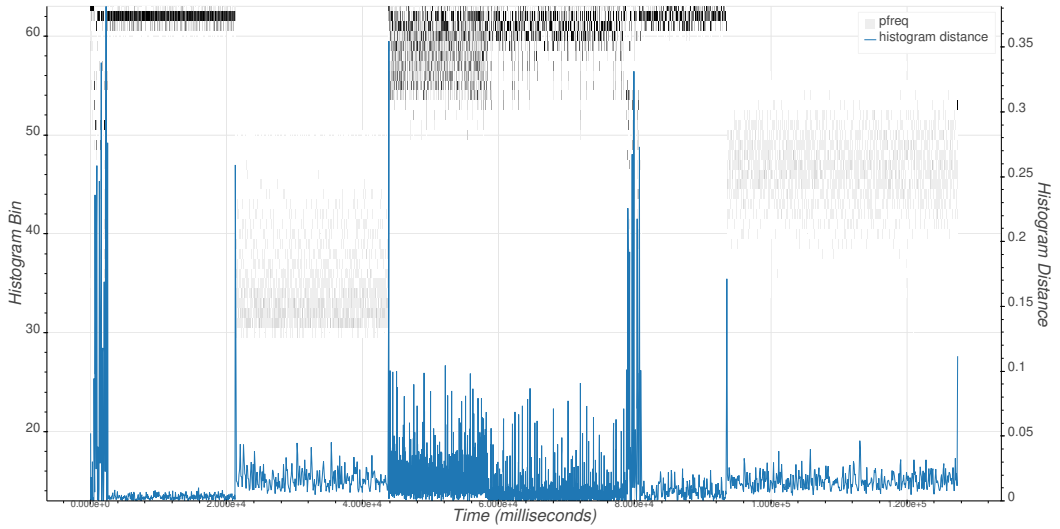
Algorithm 1 shows the pseudo-code used for measuring the distance between two histograms H_1 and H_2 of N bins. The distance represents the **minimum** number of necessary bin element movements, in order to transform H_1 to H_2 (or vice-versa). A movement is defined as the cost of moving 1 element from a bin to any of its neighbours (i.e. left or right). Starting from left to right, the $prefixSum$ variable holds the bin elements difference for all previous bins. At each iteration, the number of excessive bin elements that need to move to some bin further to the right, is added to the total distance (i.e. $histDist$ variable). Note that at the last bin, the $prefixSum$ value is equal to zero, hence to compute the normalized distance we divide by $N - 1$. The time complexity of this algorithm is $O(N)$, thus it has low-overhead and is well suited for being used online.

To further understand this procedure, Figure 7.3 shows the histogram dissimilarity that was calculated between histograms of consecutive intervals, which is visualized using the blue line. Figure 7.3a (top) shows how the core voltage values of the `bzip2` workload change over time, which are plotted using black boxes. These boxes represent the histogram bins. The indices of the bins are shown at the left y-axis. The darker the color of a box, the more sampled values belong to that bin. Note how large the dissimilarity values get, when a big shift in core voltage values occurs. We can make the same observations for the bottom figure, which presents the behavior of the productive frequency values for the `astar` workload, over time.

Since we should end up with a single value for representing the "distance" between different execution profiles, we compute the **maximum** between the core voltage and



(A) Core Voltage histograms for **bzip2** workload



(B) Productive Frequency histograms for **astar** workload

FIGURE 7.3: Histogram dissimilarity values between consecutive interval. Figure 7.3a shows the core voltage histograms for the bzip2 workload, while Figure 7.3b shows the productive frequency ones for the astar workload

productive frequency dissimilarity values, obtained from comparing the respective histograms. This allows to detect phase changes even when **only one of the two** measured metrics exhibits some variation.

7.3.4 Phase Change Detection and Classification

Now that we can efficiently quantify the dissimilarity among different execution profiles, we focus on how to exploit this information to decide when a phase change has occurred. The most simple approach is to compare the dissimilarity value with a **pre-defined fixed threshold** value. Unless the former is larger than the latter, we assume no change of program phase. In general, the larger the threshold, the fewer program phases are discovered but (usually) of longer lengths. For our experiments we set this threshold equal to **0.25**, as this value achieves a good trade-off between phase length and phase count. It is also possible to adaptively adjust this threshold, by observing the fluctuations of core voltage and productive frequency values, yet we leave this addition as future work.

Apart from detecting phase changes, we are interested in grouping similar execution profiles into a single labelled cluster, which we interpret as an individual program phase. For this purpose we utilize a leader-follower clustering scheme [18], which also allows for fast online clustering. For each cluster, we maintain its centroid, in the spirit of the popular **k-means clustering algorithm** [40].

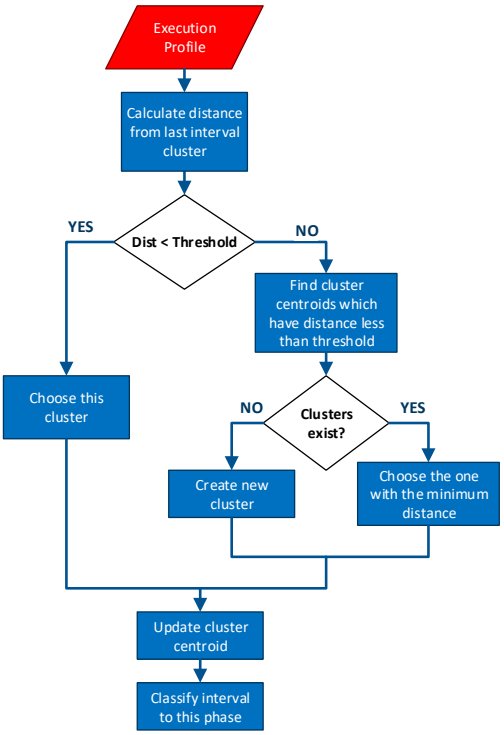


FIGURE 7.4: Phase classification algorithm

The algorithm is visualized in figure 7.4 and works as follows:

1. At first, we calculate the dissimilarity between the current execution profile and the centroid of the cluster where the previous execution profile was classified to. If the profiles are similar enough (i.e. based on the threshold value), we classify this execution profile to the same cluster / phase.

2. In the opposite case, we find all clusters, whose distance from the current execution profile is below threshold. If no such cluster exists, we create a new one

and we set its centroid to this execution profile. Otherwise, we select the one with the minimum distance and we classify this profile to that cluster.

Therefore, at the end of the program execution, we end up with multiple program phases that show similar core voltage and productive frequency characteristics.

7.4 Integrating application context

Nevertheless, there are cases when the executed code from different parts of the program results in similar core voltage and productive frequency behavior. Although this is often not an issue when detecting phase changes, classifying different code sections into the same program phase is sometimes not desirable (e.g. when optimizing for memory locality). To this end, we develop a complementary mechanism that captures application context by sampling the **instruction pointer** (IP).

We sample IPs at the same interval (i.e. $\sim 1ms$) used the rest of the metrics. IP values can be accessed from within the kernel (as described in the next section). These samples should then be combined into a single vector, which represents application execution profile during the last interval. However, since IP samples represent memory addresses, they can take huge values up to 2^{64} (for a 64-bit system).

Researchers have used Random Projection (RP) [3] techniques in order to reduce the values range. By looking at the binary representation of the addresses, they discard enough bits from random positions to obtain a smaller value on a predefined range. For example a 40-bit address can be "transformed" to a 5-bit one by dropping 35-bits, so that it can fit into a $2^5 = 32$ positions vector. Even though many different values can still end up on the same vector positions, for real-world purposes different phases can usually be distinguished.

Despite the wide utilization of Random Projection, we experiment with an ad-hoc algorithm with the end goal of minimizing the spatial information loss. Instead of dropping random bits, we partition the binary representation of the IPs into equal-sized groups and we select a single bit from each group. In particular, the algorithm performs the following steps:

1. We retrieve the first and the last address of the program `.text` section and we compute the instruction pointer range.

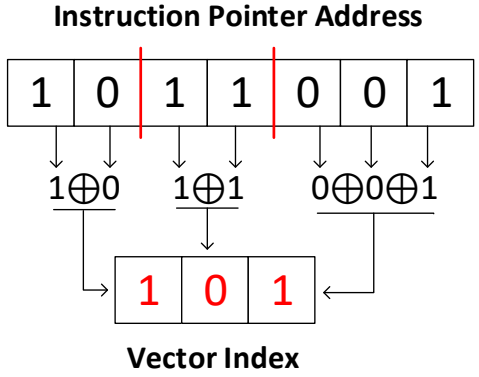


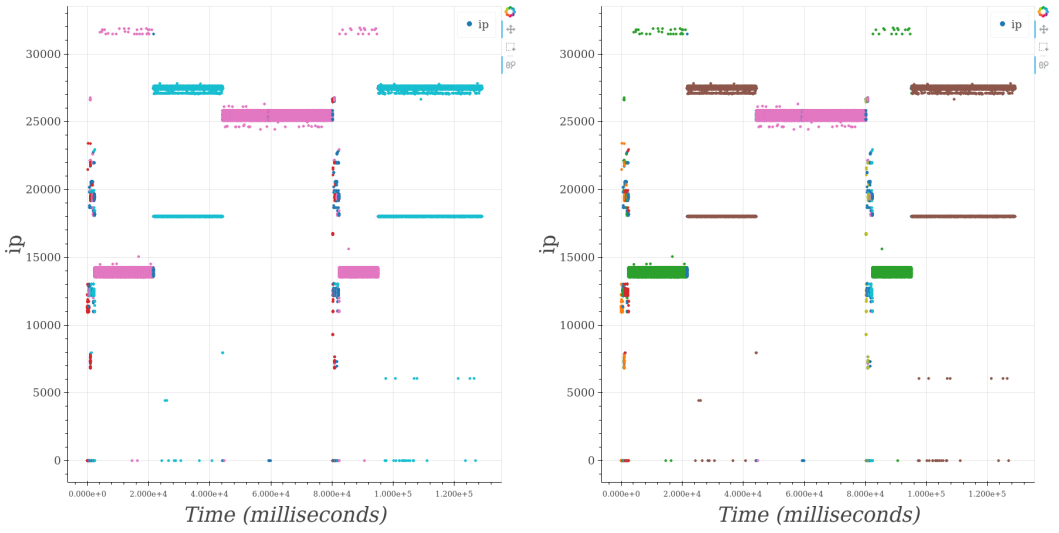
FIGURE 7.5: Spatial-aware projection

2. Depending on this range and the desired vector size, we calculate the size of each group. When possible we use equal-sized groups, otherwise bigger groups end up on the least significant bits.

3. The elements of each group are XORed together to produce a single value. Finally, these values are concatenated to derive the vector index.

Figure 7.5 illustrates the previous procedure, where a 7-bit address is reduced to a 3-bit one, which can be safely inserted at any position of a vector with $2^3 = 8$ elements.

To better understand how the integration of the application context can alter the phase classification results, Figure 7.6 presents on which phase the IP values of the `astar` workload, where classified to. Keep in mind that IP values that are close with each other, should be classified to the same phase, which is denoted by the same color in these images. On the left image, the application context is not taken into account, while or the right one, it does. We can clearly see that on the left image, the "pink" phase consists of two very different code segments, while on the right one, these segments were classified to different phases (i.e. green and pink phases).

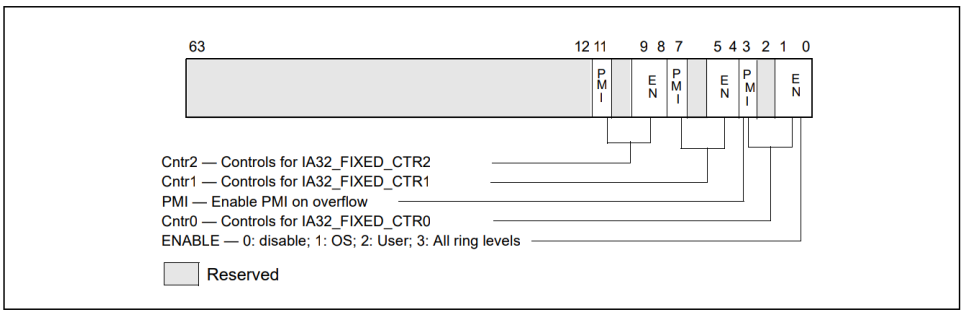


(A) Before (B) After

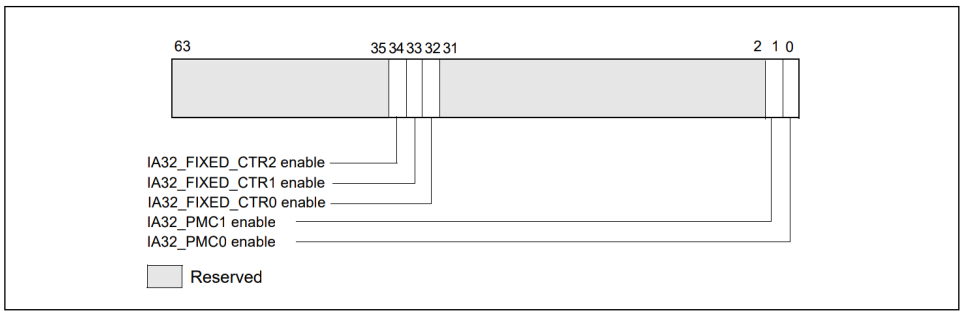
FIGURE 7.6: Classification results before (left) and after (right) integrating application context into the method, for `astar`. Same color denotes the same program phase

7.5 Implementation

Our phase detection and classification tool consists of two main components: a kernel module and a user-space component, which were developed with the goal of minimizing the overhead on the execution time of the running workload.



(A) Layout of `IA32_FIXED_CTR_CTRL` MSR



(B) Layout of `IA32_PERF_GLOBAL_CTRL` MSR

FIGURE 7.7: Layout of MSRs used by the kernel module

The kernel module is primarily responsible for collecting the actual core voltage and productive frequency measurements. For that purpose, the module spawns a kernel thread (i.e. **sampling thread**), which periodically reads these values. Note that we reused the code that was developed for extending `perf_events` (check chapter 3), where this sampling process is explained in greater detail.

In addition to these values, the program instruction pointer also needs to be retrieved. This is achieved by accessing the `regs` field of the appropriate `task_struct` structure that belongs to the monitored program. However, these values are not updated in real-time as the program executes, but rather at every context-switch. Therefore, we bind both the kernel module and the program to the same core. Now, since both programs run on the same core, a context-switch is inevitable between the two, which allows us to read the correct updated values.

Since the interval length is defined in terms of the number of executed instructions (i.e. 100M instructions), we need a mechanism that sends a notification when an interval ends. Fortunately, this functionality is provided by PMUs located on each processor core. We can configure this mechanism to raise a **Non-Maskable Interrupt** (NMI) once 100M instructions have finished executing on this core. Then, we capture this interrupt from our kernel module, and we can safely switch to the user-space component for decision making.

More specifically, the following steps are needed to correctly setup the PMUs for raising an interrupt and for counting the total retired instructions and core cycles, as

Name	Domain	Name	Domain
astar	AI / Path finding	**bwaves**	Fluid Dynamics
bzip2	Compression	**dealII**	Finite Element Analysis
gcc	C Compiler	**mcf**	Combinatorial Optimization
perlbench	Programming Language	**wrf**	Weather Forecasting
xalancbmk	XML Processing		

TABLE 7.2: Subset of SPEC CPU2006 workloads used for phase detection

documented in [31]. Please recall that these two metrics are also needed for calculating CPI, which is used for deriving the ground-truth for program phases:

1. Since the PMU sends an interrupt when the respective 64-bit counter overflows, we initially set the value of the counter that measures retired instructions (i.e. `IA32_FIXED_CTR0`) equal to $2^{64} - 100M$.

2. To enable the event counting for the fixed-function performance events only for user-space, we set bits 1 (for retired instructions) and 5 (for core cycles) on the `IA32_FIXED_CTR_CTRL` MSR (layout shown on figure 7.7a). Furthermore, in order to raise the interrupt, the 3rd bit is set as well.

3. To start the counting procedure of retired instructions and the counting of core cycles we set bits 32 and 33, respectively, of the `IA32_PERF_GLOBAL_CTRL` MSR (layout shown on figure 7.7b).

This process is repeated at the beginning of each new interval. Once the interval ends, the samples should be accessible to the user-space component for processing. This is accomplished by allocating a **shared-memory segment** inside the kernel. The user-space component can effectively gain access to this memory segment, using the `nmap` system call. The use of shared memory is one of the most efficient, in terms of performance, methods of passing data between kernel-space and userland, since no copying is performed (i.e. zero-copy).

For the most part, the user-space component is blocked and waits for the kernel module to provide a notification for a new interval. Once this happens, it checks if the program is still running, and if so performs the actual phase classification using the steps described in the previous Sections. Finally, once program execution is finished, it exports the results to a `.csv` file. Figure 7.8 illustrates the entire communication and processing procedure for both components.

7.6 Evaluation

In this Section we evaluate the performance of our phase classification tool (i.e. VPF) in terms of **phase homogeneity**. We use a subset of workloads from the SPEC

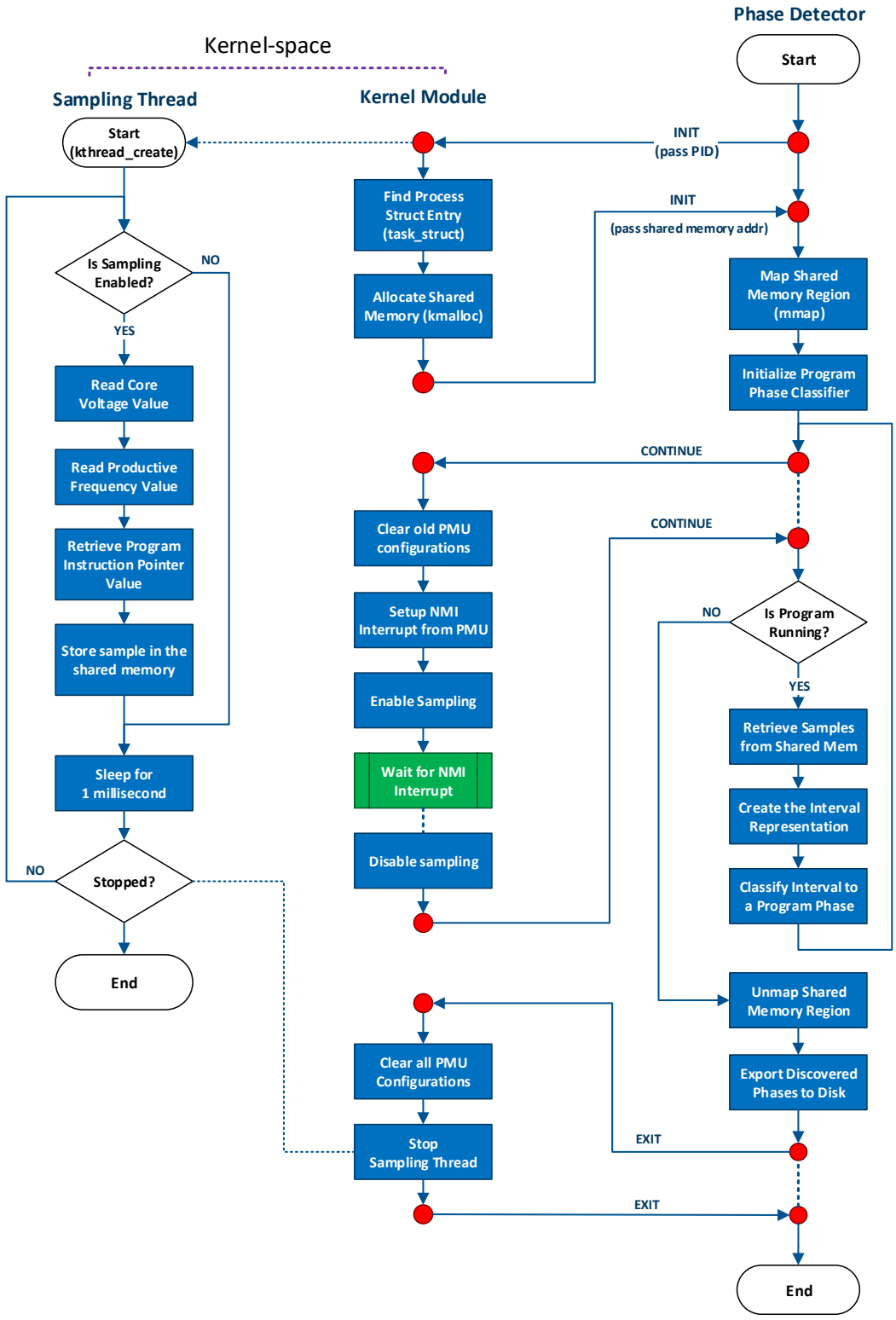


FIGURE 7.8: Workflow of our online phase detection tool

`CPU2006`, which have shown the most interesting program phase behavior. Table 7.2 lists these workloads.

In order to quantify phase homogeneity, we calculate the CPI variance of all samples that were classified in the same phase. For this purpose, we use the Coefficient of Variation (CoV) [57] that measures the dispersion of a distribution (CPI distribution in our case):

$$c_v = \frac{\sigma}{\mu}$$

where σ is the standard deviation and μ is the mean value of the samples, respectively. For each program phase we compute the CoV and then we take the average across all phases. If a program phase consists of a single interval, then we penalize this phase by setting its CoV equal to the CoV of the entire program execution (i.e. CPI variation across all samples).

We compare the performance of our tool, with (i.e. VPF-IP) or without (i.e. VPF) integrating the monitored program application context, against the state-of-the-art [54] `ScarPhase`. The phase detection threshold was set to 0.25 in our case and to 0.45 for ScarPhase, to approximately match the number of detected phases.

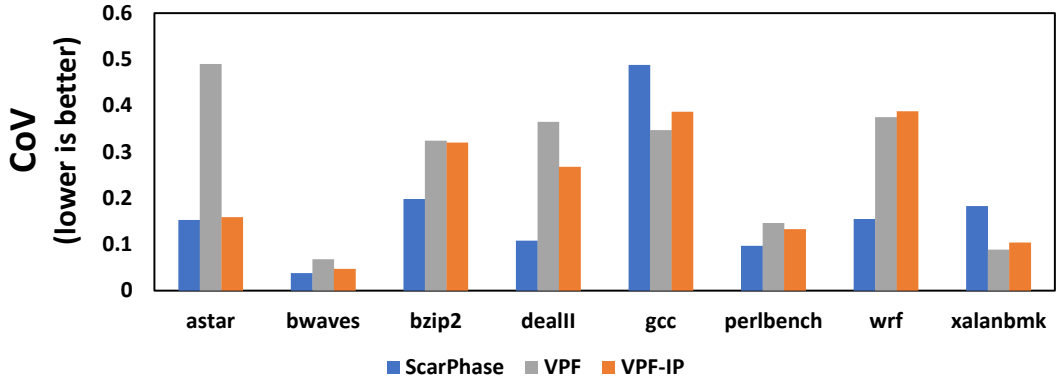


FIGURE 7.9: Homogeneity comparison of program phases discovered

Figure 7.9 graphically shows the results for all tools and configurations. For `bzip2`, `dealII` and `wrf` ScarPhase performs better, while for `gcc` and `xalanbmk` VPF does. Both tools achieve the same level of phase homogeneity for `bwaves` and `perlbench`. An interesting case is the `astar` workload, where the integration of the application context led to a huge improvement that managed to match ScarPhase performance. By taking a closer look on Figure 7.3b at around 60 seconds, we can see the reason behind this improvement. Even though, we can see that the between 44 and 60 seconds the productive frequency distribution is more spread out, while between 60 and 80 seconds is narrower, yet the histogram distance line never exceeds the 0.25 threshold, and thus failing at detecting a phase change.

Finally, we compared the runtime overhead of both tools. On the same configuration (i.e. ScarPhase threshold equal to 0.45, while VPF threshold equal to 0.25), ScarPhase has a **2.5**% runtime overhead, which translates to 25ms per second. For the VPF, we measured both the kernel module and the user-space component overhead. The overhead of the former is $\sim$ 0.25% and the one of the latter is $\sim$ 0.9%, for a total of **1.15**% or 11.5ms per second. Thus the runtime overhead of VPF is **less than half** compared with that of ScarPhase.

Chapter 8

Conclusions

In this thesis, firstly, we explored the relationship between profiled hardware events that model the workload behavior with the CPU core voltage values, as these are adjusted by the `SpeedShift` DVFS mechanism on the Skylake architecture. Using correlation analysis we showed that different workloads can have widely different impact on the strength and the direction of the correlation. Surprisingly, we found that for the same hardware event the correlation direction can be both positive or negative, depending on the actual workload.

Next, we showed that it is possible to use machine learning models to get satisfying estimations on the CPU core voltage of a workload, at any time point. This holds true even when training the models using solely performance events of the same workload. However, due to the diversity of the workload behavior, it was not possible to achieve decent accuracy on unseen workloads. Furthermore, we observed that the utilization of hardware event values from previous intervals, did not lead to significant accuracy gains. On possible direction for future work, is to investigate the performance of online machine learning methods on unseen workloads.

Lastly, we showed that CPU core voltage and productive frequency values can reveal a great deal of information regarding the temporal architectural phases of a program. Moreover, the integration of application context to account for different code phases, resulted in better phase for the majority of our workloads. Since we design and developed our tool based the assumption of serial workloads, one could try to generalize our approach and experiment with multi-threaded workloads.

Bibliography

[1] Rajeev Balasubramonian et al. "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures". In: *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. ACM. 2000, pp. 245–257.

[2] Natalia Becker et al. "penalizedSVM: a R-package for feature selection SVM classification". In: *Bioinformatics* 25.13 (2009), pp. 1711–1712.

[3] Ella Bingham and Heikki Mannila. "Random projection in dimensionality reduction: applications to image and text data". In: *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2001, pp. 245–250.

[4] Leo Breiman. *Classification and regression trees*. Routledge, 2017.

[5] Kenneth P Burnham and David R Anderson. *Model selection and multimodel inference: a practical information-theoretic approach*. Springer Science & Business Media, 2003.

[6] Gavin C Cawley and Nicola LC Talbot. "On over-fitting in model selection and subsequent selection bias in performance evaluation". In: *Journal of Machine Learning Research* 11.Jul (2010), pp. 2079–2107.

[7] Daniele Cesarini, Andrea Bartolini, and Luca Benini. "Benefits in Relaxing the Power Capping Constraint". In: *Proceedings of the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy Efficient HPC Systems*. ANDARE '17. Portland, OR, USA: ACM, 2017, 3:1–3:6. ISBN: 978-1-4503-5363-2. DOI: 10.1145/3152821.3152878. URL: http://doi.acm.org/10.1145/3152821.3152878.

[8] Sung-Hyuk Chaa and Sargur N Sriharib. "On measuring the distance between histograms". In: *Pattern Recognition* 35 (2002), pp. 1355–1370.

[9] Nitesh V Chawla et al. "SMOTE: synthetic minority over-sampling technique". In: *Journal of artificial intelligence research* 16 (2002), pp. 321–357.

[10] G. Contreras and M. Martonosi. "Power prediction for Intel XScale/spl reg/ processors using performance monitoring unit events". In: *ISLPED '05. Proceedings of the 2005 International Symposium on Low Power Electronics and Design, 2005.* 2005, pp. 221–226. DOI: 10.1109/LPE.2005.195518.

[11] George Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[12] Bob Davies et al. "ipart: An automated phase analysis and recognition tool". In: *Technical Report IR-TR-2004-1-iPART, Intel Corporation* (2004).

[13] R. H. Dennard et al. "Design Of Ion-implanted MOSFET's with Very Small Physical Dimensions". In: *Proceedings of the IEEE* 87.4 (1999), pp. 668–678. ISSN: 0018-9219. DOI: 10.1109/JPROC.1999.752522.

[14] Susan J. Devlin, R. Gnanadesikan, and J. R. Kettenring. "Robust Estimation and Outlier Detection with Correlation Coefficients". In: *Biometrika* 62.3 (1975), pp. 531–545. ISSN: 00063444. URL: http://www.jstor.org/stable/2335508.

[15] Ashutosh S Dhodapkar and James E Smith. "Comparing program phase detection techniques". In: *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2003, p. 217.

[16] Y. Dodge. *The Concise Encyclopedia of Statistics*. The Concise Encyclopedia of Statistics. Springer New York, 2008. ISBN: 9780387317427.

[17] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2121–2159.

[18] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012.

[19] R. Efraim et al. "Energy Aware Race to Halt: A Down to EARtH Approach for Platform Energy Management". In: *IEEE Computer Architecture Letters* 13.1 (2014), pp. 25–28. ISSN: 1556-6056. DOI: 10.1109/L-CA.2012.32.

[20] M. A. Efroymson. "Multiple Regression Analysis". In: *Mathematical Methods for Digital Computers*. Ed. by A. Ralston and Eds H. S. Wilf. New York: John Wiley, 1960.

[21] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*. Vol. 1. 10.

[22] Karl Pearson F.R.S. "LIII. On lines and planes of closest fit to systems of points in space". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11 (1901), pp. 559–572. DOI: 10.1080/14786440109462720. eprint: https://doi.org/10.1080/14786440109462720. URL: https://doi.org/10.1080/14786440109462720.

[23] Marcel van Gerven and Sander Bohte. *Artificial neural networks as models of neural information processing*. Frontiers Media SA, 2018.

[24] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 315–323.

[25] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[26] John L. Henning. "SPEC CPU2006 Benchmark Descriptions". In: *SIGARCH Comput. Archit. News* 34.4 (Sept. 2006), pp. 1–17. ISSN: 0163-5964. DOI: 10.1145/1186736.1186737. URL: http://doi.acm.org/10.1145/1186736.1186737.

[27] Tin Kam Ho. "Random decision forests". In: *Proceedings of 3rd International Conference on Document Analysis and Recognition*. Vol. 1. 1995, 278–282 vol.1. DOI: 10.1109/ICDAR.1995.598994.

[28] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: 9 (Dec. 1997), pp. 1735–80.

[29] *Intel VTune Performance Analyzer*. URL: https://software.intel.com/en-us/vtune.

[30] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D*. URL: https://software.intel.com/en-us/articles/intel-sdm.

[31] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*. URL: https://software.intel.com/en-us/articles/intel-sdm.

[32] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 4: Model-Specific Registers*. URL: https://software.intel.com/en-us/articles/intel-sdm.

[33] *Intel® Turbo Boost Technology 2.0*. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.

[34] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[35] Gareth James et al. *An introduction to statistical learning*. Vol. 112. Springer, 2013.

[36] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[37] LWN.net. *Perfcounters added to the mainline*. URL: https://lwn.net/Articles/339361/.

[38] LWN.net. *RAPL (Running Average Power Limit) driver*. URL: https://lwn.net/Articles/545745/.

[39] LWN.net. *Scaled statistics using APERF/MPERF in x86*. URL: https://lwn.net/Articles/283769/.

[40] James MacQueen et al. "Some methods for classification and analysis of multivariate observations". In: 1967.

[41] Deborah T. Marr et al. "Hyper-Threading Technology Architecture and Microarchitecture." In: *Intel Technology Journal* 6.1 (2002), p. 1. ISSN: 1535864X. URL: http://search.ebscohost.com/login.aspx?direct=true&db=bsx&AN=6769155&site=eds-live.

[42] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012. ISBN: 026201825X, 9780262018258.

[43] Konstantin Moiseev, Avinoam Kolodny, and Shmuel Wimer. "Timing-aware Power-optimal Ordering of Signals". In: *ACM Trans. Des. Autom. Electron. Syst.* 13.4 (Oct. 2008), 65:1–65:17. ISSN: 1084-4309. DOI: 10.1145/1391962.1391973. URL: http://doi.acm.org/10.1145/1391962.1391973.

[44] Noor Mubeen. "Workload Frequency Scaling Law - Derivation and Verification". In: *Queue* 16.2 (Apr. 2018), 50:50–50:66. ISSN: 1542-7730. DOI: 10.1145/3190560. URL: http://doi.acm.org/10.1145/3190560.

[45] Andrew Y. Ng. "Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance". In: *Proceedings of the Twenty-first International Conference on Machine Learning*. ICML '04. Banff, Alberta, Canada: ACM, 2004, pp. 78–. ISBN: 1-58113-838-5. DOI: 10.1145/1015330.1015435. URL: http://doi.acm.org/10.1145/1015330.1015435.

[46] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training recurrent neural networks". In: *International Conference on Machine Learning*. 2013, pp. 1310–1318.

[47] Nitzan Peleg and Bilha Mendelson. "Detecting Change in Program Behavior for Adaptive Optimization". In: *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. PACT '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 150–162. ISBN: 0-7695-2944-5. DOI: 10.1109/PACT.2007.25. URL: https://doi.org/10.1109/PACT.2007.25.

[48] J. Ross Quinlan. "Induction of decision trees". In: *Machine learning* 1.1 (1986), pp. 81–106.

[49] Herbert Robbins and Sutton Monro. "A stochastic approximation method". In: *Herbert Robbins Selected Papers*. Springer, 1985, pp. 102–109.

[50] E. Rotem. *Intel® Architecture, Code Name Skylake Deep Dive: A New Architecture to Manage Power Performance and Energy Efficiency*. Intel Developer Forum, Aug 2015.

[51] Efi Rotem et al. "Analysis of Thermal Monitor Features of the Intel Pentium M Processor". In: *in Workshop on Temperatureaware Computer Systems*. 2004.

[52] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[53] Philip Sedgwick. "Pearson's correlation coefficient". In: *BMJ* 345 (2012). DOI: 10.1136/bmj.e4483. URL: https://www.bmj.com/content/345/bmj.e4483.

[54] Andreas Sembrant, David Eklov, and Erik Hagersten. "Efficient software-based online phase classification". In: *Workload Characterization (IISWC), 2011 IEEE International Symposium on*. IEEE. 2011, pp. 104–115.

[55] Xipeng Shen, Yutao Zhong, and Chen Ding. "Locality phase prediction". In: *ACM SIGPLAN Notices* 39.11 (2004), pp. 165–176.

[56] T. Sherwood, E. Perelman, and B. Calder. "Basic block distribution analysis to find periodic behavior and simulation points in applications". In: *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. 2001, pp. 3–14. DOI: 10.1109/PACT.2001.953283.

[57] Timothy Sherwood, Suleyman Sair, and Brad Calder. "Phase Tracking and Prediction". In: *Proceedings of the 30th Annual International Symposium on Computer Architecture*. ISCA '03. San Diego, California: ACM, 2003, pp. 336–349.

ISBN: 0-7695-1945-8. DOI: 10.1145/859618.859657. URL: http://doi.acm.org/10.1145/859618.859657.

[58] Timothy Sherwood et al. "Automatically Characterizing Large Scale Program Behavior". In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS X. San Jose, California: ACM, 2002, pp. 45–57. ISBN: 1-58113-574-2. DOI: 10.1145/605397.605403. URL: http://doi.acm.org/10.1145/605397.605403.

[59] Karan Singh, Major Bhadauria, and Sally A. McKee. "Real Time Power Estimation and Thread Scheduling via Performance Counters". In: *SIGARCH Comput. Archit. News* 37.2 (July 2009), pp. 46–55. ISSN: 0163-5964. DOI: 10.1145/1577129.1577137. URL: http://doi.acm.org/10.1145/1577129.1577137.

[60] Tyler Sondag and Hridesh Rajan. "Phase-based tuning for better utilization of performance-asymmetric multicore processors". In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2011, pp. 11–20.

[61] Dan Terpstra et al. "Collecting Performance Data with PAPI-C". In: *Tools for High Performance Computing 2009*. Ed. by Matthias S. Müller et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173.

[62] Robert Tibshirani. "Regression Shrinkage and Selection via the Lasso". In: *Journal of the Royal Statistical Society. Series B (Methodological)* 58.1 (1996), pp. 267–288. ISSN: 00359246. URL: http://www.jstor.org/stable/2346178.

[63] Vincent Weaver. "System-wide Performance Counter Measurements: Offcore, Uncore, and Northbridge Performance Events in Modern Processors". In: 2017.

[64] Arnold D Well and Jerome L Myers. *Research design & statistical analysis*. Psychology Press, 2003.

[65] Arch Linux Wiki. *CPU frequency scaling*. URL: https://wiki.archlinux.org/index.php/CPU_frequency_scaling.

[66] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785.

[67] Shaomin Wu. "A review on coarse warranty data and analysis". In: *Reliability Engineering and System Safety* 114 (2013), pp. 1 –11. ISSN: 0951-8320. DOI: https://doi.org/10.1016/j.ress.2012.12.021.

[68] A. Yasin. "A Top-Down method for performance analysis and counters architecture". In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2014, pp. 35–44. DOI: 10.1109/ISPASS.2014.6844459.

[69] Hui Zou and Trevor Hastie. "Regularization and variable selection via the Elastic Net". In: *Journal of the Royal Statistical Society, Series B* 67 (2005), pp. 301–320.