DISCLAIMER:

This document does not meet the
current format guidelines of
the Graduate School at
The University of Texas at Austin.

It has been published for
informational use only.

Copyright

by

Erika Susana Alcorta

2019

The Report Committee for Erika Susana Alcorta
Certifies that this is the approved version of the following Report:

Architectural exploration of a low-complexity machine learning model for dynamic performance forecasting

APPROVED BY
SUPERVISING COMMITTEE:

Andreas Gerstlauer, Supervisor

Aswin Ramachandran

Architectural exploration of a low-complexity machine learning model for dynamic performance forecasting

by

Susana Alcorta

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2019

Dedicated to my parents, to whom I owe my life and achievements.

Acknowledgments

I wish to thank the people who helped me: my supervisor, Dr. Andreas Gerstlauer, for his patience and valuable feedback; my mentor, Dr. Aswin Ramachandran, for his lessons, and availability. I also thank my colleagues, Alexander Cathis, Sabine Francis, and Erin Blake, for their proofreading and suggestions that made my seemingly endless writing time easier. And I thank Intel, who provided many of the tools to make this project possible. I also wish to display my gratitude to my sponsor, Fulbright, for giving me the opportunity of studying my Master's degree. Finally, I thank my family for their complete support all this time.

Architectural exploration of a low-complexity machine learning model for dynamic performance forecasting

Susana Alcorta, M.S.E.
The University of Texas at Austin, 2019

Supervisor: Andreas Gerstlauer

A computer system that supports frequency scaling cannot be guaranteed an optimal frequency selection when it does not have information about the workloads that it is running. Algorithms in previous work either select the best frequency for the current period only reacting to changes, whereas others aim at predicting the future state to select the best frequency for the following period.

This project investigates the latter type of algorithms and proposes two machine learning-based models to predict the performance of a workload at runtime. We first motivate the importance of predicting changes and demonstrate that this is a challenging problem because they are very infrequent. Previous work has used table-based predictors of phases, but we eliminate the use of tables and frame our problem as a time series forecasting problem. Various approaches have been proposed for time series forecasting over the years. Most recently, machine learning based approaches have shown superior results.

Our models are based on a machine learning structure called long short-term memory (LSTM) that can process time-dependent data. One of our models is trained by feeding its LSTM with the consecutive observations from the traces to perform continuous forecasting. The other model compresses them first and uses run-length encoding (RLE) to train the LSTM to find workload changes.

We apply our approach for workload prediction on state-of-the-art computer platforms. We generate traces by periodically collecting data from the performance monitoring units of an Intel 10th-generation platform. We split traces into training and test sets for learning and prediction. Results show that despite changes being infrequent, a continuous forecasting model performs better than a RLE-based one. Compared to an approach that simply predicts the current value for the next period, continuous forecasting can predict 43% of the performance changes while a RLE-based model only predicts 14%. Overall, continuous and RLE-based forecasting decrease the forecasting error by 7% and 5% compared to same-value prediction.

Chapter 1

Introduction

Dynamic voltage and frequency scaling (DVFS) is a technique in computer architecture that allows controlling the trade-offs between power consumption and computational performance [3]. Said trade-offs are navigated in modern processors by making voltage-frequency (V/f) pair selections per core. However, obtaining the optimal V/f point of operation of all cores in a multi-core environment is still an open question [20]. Furthermore, as opposed to real-time systems where the workload, arrival time, and deadline of tasks are known, the selection of optimal V/f of a single core on a general-purpose machine cannot be guaranteed when there is no information given about the future behavior of the workload [5, 11]. Given a power/performance trade-off (e.g., maximize performance per Watt) and a power budget, many algorithms have been proposed to solve these questions [21, 20, 19, 18]. Some of them read the state of the system and select a V/f pair based on a given policy; others are more sophisticated in recording workload behavior trends to predict the best selection.

Previous work has characterized program phases with performance metrics such as cycles per instruction (CPI) [22] and built table-based predictors

to estimate future phases [4]. Similarly, the goal of this project is to accurately predict what the next performance metric will be at each DVFS sampling period, but with a different approach: the problem of performance prediction is framed as a time series forecasting problem. A time series is defined as a chronological sequence of observations on a variable of interest [23]. The data collected periodically to measure the performance of the system forms a time series, and this work explores models to forecast new values.

Artificial neural networks (ANN) have been popular in solving many forecasting problems [12, 26]. The resulting models can be very complex in terms of computation and storage. However, we find that a popular structure for handling time-dependent data, the long short-term memory (LSTM) [14], can be very accurate while minimizing complexity. This project explores several architectures to use an LSTM-based algorithm that periodically predicts dynamic workload behaviors that are useful inputs for DVFS management.

The contributions of this project report are summarized as follows:

1. We collect performance counters and perform a study of the periodic behavior of some benchmarks from the SPECspeed 2017 Integer and Floating-Point suites

2. We evaluate the accuracy and complexity of different LSTM models and their parameters in predicting core CPI

3. We present how the accuracy of LSTM can be improved by varying the input performance counters and their formatting

4. We propose the use of run-length encoding to reduce the memorization overhead of the LSTM models

The rest of this report is organized as follows: previous work in the areas of DVFS, performance prediction, and time series are summarized in Chapter 2. The background of the baseline and the models evaluated in Chapter 6 is presented in Chapter 3. Chapter 4 defines the problem statement and presents the proposed solutions. Chapter 5 describes the methodology for data collection and the framework used for evaluation. Finally, Chapter 7 discusses the conclusion and future work.

Chapter 2

Related Work

2.1 Dynamic Voltage and Frequency Scaling

DVFS mechanisms operate periodically. In each period, measurements are collected, and a V/f pair is output. We make a distinction between **reactive** and **predictive** mechanisms. The inputs of **reactive** mechanisms are the observations sampled at the current period only, and they optimize the V/f for the current period. On the other hand, **predictive** mechanisms estimate the observation of the next period(s) and optimize the V/f selection accordingly. Looking ahead may require storing some history, so **predictive** mechanisms may use more inputs other than the current observations.

The DVFS module of PGCapping, by Ma and Wang [21], is an example of a **reactive** mechanism. The inputs include current measurements of power and performance (in terms of utilization) of each core and the operating V/f pairs. They iterate a V/f table and use the values to estimate the change in power and performance to greedily select the best power-reduction to performance-loss ratio. Cochran et al. [7] develop a classifier of counters from the performance monitoring units (PMU) that calculates the probability for each V/f pair of yielding an optimal workload performance. The classifier

is trained offline and implemented online. The implementation samples PMU events at each DVFS period and feeds them into the classifier. We consider this is a **reactive** mechanism as well because the classifier takes as inputs the present PMU measurements.

Notice that **reactive** algorithms assume that the sampled state of the system will remain the same for the next period. They do not account for the changes that may occur in the workloads. This project aims at improving the decisions of such mechanisms by using predictions for the next period as their input. Using PMU counters is a popular technique for DVFS [11, 7, 18]; thus, this work focuses on sampling and predicting PMU events.

2.2 Phase Prediction

Phase classification and prediction is a line of research that aims at foretelling the behaviors of a workload at running time. Roughly speaking, a phase is represented by intervals of time that have similar performance metrics. After detecting and classifying them, a predictor can be used for multiple purposes, such as estimating phase duration [27] or anticipating what the next phase will be [18]. The predictions of these mechanisms are used for DVFS; hence, they fall under our **predictive** classification of DVFS management.

An online phase classifier can be a costly approach. Cruz et al. [10] look at and extend the branch target buffer to find loops. After one is encountered multiple times, it is classified as a new phase. This approach uses various tables resulting in a hardware overhead of 8.2MB.

Chang et al. [4] proposed a phase classifier that samples a thread's extended instruction pointer (EIP) every one million instructions. They use a history table to track the number of times that different combinations of EIP counters have been encountered in 100 samples, and each one is labeled as a phase. Then, they use Markov decision tables and run-length encoding to predict the next label with up to 80% accuracy. Our project also uses run-length encoding but eliminates the use of tables.

The work by Srinivasan et al. [27] is an example of how to apply phase classification to DVFS. They use a linear adaptive filter to predict the duration of classified phases. To characterize them, they use memory access rate (MAR) and instructions per cycle (IPC). The DVFS regulator is only triggered when the predicted duration is longer than a threshold. They report a misprediction rate of less than 12%. Their predictions include only the length of a phase, while this work also aims at predicting what the following one will be.

2.3 Time Series Forecasting

Time series forecasting has been used in a variety of fields such as economics, social sciences, and medicine [23]. Each problem is different, and there is not a single model that works for all. Thus, multiple time series analysis tools exist. The auto-regressive integral moving average (ARIMA) is a canonical forecasting model found in textbooks. The exponentially weighted moving average (EWMA) is introduced as a data smoother but has also been used for forecasting [25]. It is not as popular as ARIMA, but we find that for

this project, it involves lower training and implementation complexity. The details about how it works are presented in Chapter 3.

Time series forecasting has not been the exception to the widely spreading range of applications of machine learning. Recent work by Selvin et al. [26] shows how different machine learning models outperform ARIMA in the prediction of stock prices. LSTM is one of those models. It has been used in traditional forecasting problems such as traffic, weather, and stock price prediction with favorable results. Zhao et al. [29] use LSTM and compare it to other representative forecasting models for short-term traffic prediction. They estimate future traffic flow in different periods and locations. Their results show that LSTM is the model with the least relative error in the majority of their experiments. Nelson et al. [24] train an LSTM classifier to predict whether the price of a stock will increase, and use it to create a trading strategy. They apply this model to different commodities, and their results show that the LSTM has the highest accuracy as well as average return.

The data collected by DVFS mechanisms can be formulated as a time series. Given the success of forecasting with LSTM in other fields, we explore using it for DVFS applications.

Chapter 3

Forecasting Models

This chapter provides a formal description of the models that are studied in this report. We first summarize the notation for time series analysis from Montgomery et al. [23]. Then, the same-value prediction is introduced as a baseline and as a motivation of the challenges that other predictors face. Next, we present EWMA as a model to verify the value of the work proposed in this project. We choose EWMA for its low-complexity and exploration simplicity and describe its intuition in this chapter. Finally, the last section summarizes the definition of LSTM and discusses its advantages over other machine learning models.

3.1 Time Series Forecasting Notation

Suppose we have a trace of data collected periodically from the same variable. Let T be the size of our trace. We denote the observation at time period t as y_t, where $t = 1, 2, ..., T$. A forecast or predicted value of y_t that was made at some previous time, $t - \tau$, is typically denoted as $\hat{y}_t(t - \tau)$. All the predictors in this work use a value of $\tau = 1$; to simplify the notation, we will represent $\hat{y}_t(t - 1)$ as $\hat{y}_t$.

A time series can be multivariate, in which case the vector of the form $Y_t = (y_{1t}, y_{2t}, ..., y_{mt})$ consists of m univariate time series. Some of the predictors used in this project take multivariate inputs; however, all of them forecast only one variable of interest. To differentiate the variable that is forecasted from the ones that are inputs only, we use the notation y_t and x_{kt} respectively, where $k < m$ is used to index more than one variable in the multivariate time series.

3.2 Same-value Prediction

The purest form of prediction is to assume that the value for the next sampling period is equal to the current sample, $\hat{y}_{t+1} = y_t$. The DVFS algorithms that we classify as **reactive** in the previous chapter are behaving as same-value predictors.

A study was performed to investigate how often changes in the CPI are encountered in different benchmarks. The details about the data collection are described in Chapter 5. In this project, a change in CPI is defined as an increase or decrease by at least 10% of the measured value of two consecutive samples: $change_t = \frac{|y_t - y_{t-1}|}{y_{t-1}} \geq 0.1$. Figure 3.1 shows a heat map of the change versus no change fractions for each benchmark and each sampling period.

Even though the same-value prediction is a straightforward approach, it is already giving a minimum of 46% accuracy (perlbench@10ms, Fig. 3.1). It represents a challenge and a lower-bound for other prediction models, which are not as simple; the increase in complexity of a predictor should be justified

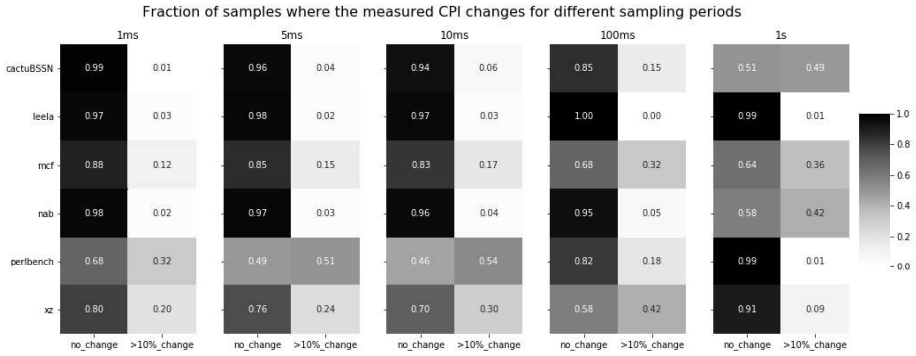

Figure 3.1: Fraction of times the CPI changes more than 10% in the traces of different sampling periods

by improving the accuracy of same-value prediction. The room of opportunity sits in being able to predict workload changes. Same-value prediction is used as the baseline of this work.

3.3 Exponentially Weighted Moving Average (EWMA)

Averaging is a technique in time series analysis used to smooth data [23]. Smoothing is required under the assumption that a trace consists of two components: signal and noise. The signal is the pattern that we are trying to recognize and be able to predict. The noise is assumed to be present as we do not have tools that can collect perfect samples. For example, in the case when performance counters values are periodically accessed from a core running a target workload, the samples may include some noise from interrupts that might run on the same core, or from resources that are shared with other cores such as caches or memory bandwidth.

Smoothers can also be used to forecast the signal. One simple form of smoothing is to average all the samples that have been observed up to time t. However, this smoother assumes that the mean of the signal is constant over time. When we know that the signal is changing, moving averages may be more appropriate. A simple moving average consists of the average of a span with size N in our data [23]:

$$M_t = \frac{y_t + yt - 1 + ... + y_{t-N+1}}{N} \tag{3.1}$$

This moving average is giving each observation in the span the same weight, $1/N$. If the importance of each observation is known, different weights, w_i, can be given to all $i = 1, 2, ..., N$. When each observation in a moving average is given different weight, and the sum of all weights is equal to 1, it is called a weighted moving average (WMA).

The vector of weights in a WMA can have any shape. If they they decay exponentially by a factor θ, $|\theta| < 1$, the approach is commonly known as EWMA. In this case, the weights need to be adjusted so that they meet the condition of summing up to 1 [23]:

$$\hat{y}_T = (1 - \theta) \sum_{t=1}^{T-1} \theta^t y_{T-t} \tag{3.2}$$

Notice that the whole trace was used to compute the EWMA. However, storing the entire history is not necessary because this equation has a recursive form [23], which when used for a forecaster looks like this:

$$\hat{y}_{t+1} = (1 - \theta) y_t + \theta \hat{y}_{t-1} \tag{3.3}$$

The recursive form of EWMA makes it easy to implement, and as such, we explore it in this project. It only requires one parameter, $\alpha = 1 - \theta$.

3.4 Predictions with Machine Learning

The output of a feed-forward ANN, such as a multi-layer perceptron (MLP) or a convolutional neural network (CNN) is not time-dependent. In other words, there is no history of past inputs or outputs to the network that can change the inference of the present output. Thus, sliding window approaches have been used [26] to implement ANN for time series forecasting. Similar to the simple moving average, a sliding window consists of a span of past observations with fixed size N. Figure 3.2 shows a graphical representation of an MLP with two hidden layers of arbitrary size. Each arrow represents a weight in the network. Notice that if the size of the sliding window increases, the number of added weights increases proportionally to the size of the first hidden layer.

For time-dependent problems, recurrent neural networks (RNN) offer a more natural solution because the output of an inference depends on the sequence of previously observed inputs. Thus, it is not necessary to explicitly feed multiple past observations as inputs to an RNN. Consequently, the number of weights that have to be stored could be reduced as compared to a feed-forward ANN.

RNN suffer from training issues that feed-forward networks do not. Because the neurons need to be unfolded for backpropagation through time

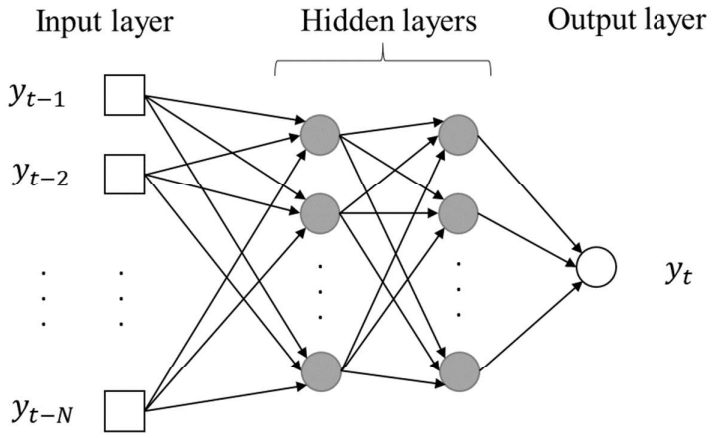

Figure 3.2: Example of sliding window approach for an ANN with univariate timeseries and hidden layers of arbitrary size

(BPTT) and the same weights are multiplied to themselves for as many times as unfolded timesteps, they suffer from exploding or vanishing gradients when their values are too large or too small. A structure called long short-term memory (LSTM) is an RNN that overcomes these issues.

An LSTM cell is composed of a memory unit and gates, which are typically named input, output, and forget. These gates can be thought of as standard neurons of a feed-forward network because they also consist of a dot product of inputs and weights and an activation function. A black box representation of an LSTM cell is shown in Figure 3.3. Equation 3.4 describes the dynamics of the math inside of it.

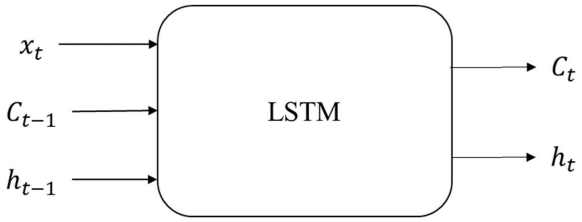

Figure 3.3: LSTM black box

$$
\begin{aligned}
i_t &= \sigma(U_i h_{t-1} + W_i x_t + b_i) \\
f_t &= \sigma(U_f h_{t-1} + W_f x_t + b_f) \\
o_t &= \sigma(U_o h_{t-1} + W_o x_t + b_o) \\
\tilde{c} &= tanh(U_c h_{t-1} + W_c x_t + b_c) \\
c_t &= f_t * c_{t-1} + i_t * \tilde{c} \\
h_t &= o_t * tanh(c_t)
\end{aligned}
\tag{3.4}
$$

The values of the input, forget, and output gates at period t are represented as i_t, f_t, and o_t, respectively. The first one controls how much the input contributes to the value stored in the memory cell, C, and the second how much of the cell's previous value to keep or forget. After the cell is updated, the output gate determines how much of its value meshes the output, h_t.

The input x_t is either a scalar value or a vector. The matrix of weights W increases proportionally to the size of the input vector, x; and, the matrix

of weights U, to the number of LSTM cells in the network.

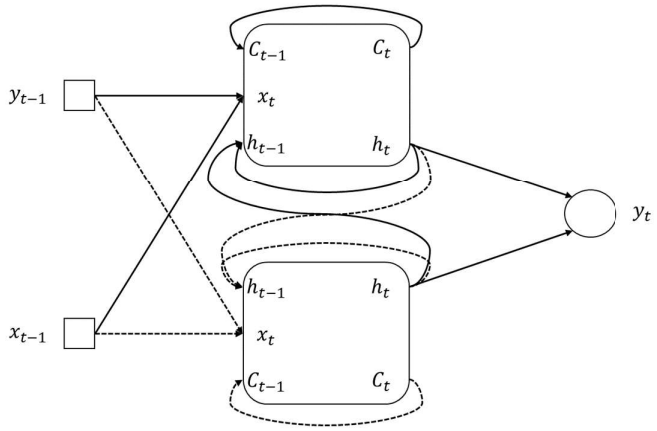

Figure 3.4: Example of an LSTM network with two inputs and two cells. The inputs of the cell below are represented as dashed arrows only to ease visualization

It is a common practice to use a dense layer as the output layer of an LSTM model. This approach is also followed in the models that are showcased in the next chapters. An example is shown in Figure 3.4. Consider a multivariate time series where we collected two variables, x and y, and our variable of interest is y. The model in Figure 3.4 is an example of how an LSTM network with two cells would look like to forecast this time series.

Chapter 4

Dynamic Performance Forecasting

Chapter 2 defined the differences between **reactive** and **predictive** DVFS mechanisms. It is relevant to point out that **predictive** mechanisms consist of broadly two steps: (1) prediction and (2) V/f selection. Our project focuses on exploring models that perform the tasks of the first step. We call the process dynamic performance forecasting, and we describe it in this chapter. Then, we present two solutions based on LSTM.

The DVFS mechanism operates periodically. At the end of each period, some data is sampled. The data represents the state of the system during that time, in some form. A **predictive** mechanism manipulates the data to predict the next state. Then, the V/f selection step decides whether to scale the frequency based on the prediction. Usually, the data is a performance measure or a representation of it. For example, phase predictors foretell labels of phases, and phases are defined by performance metrics such as IPC or MAR [27]. We define dynamic performance forecasting as the process of periodically predicting the value of the next performance measure of a workload at running-time.

Like in any other forecasting problem, the goal of a model or predictor

is to reduce the forecast error, $e_t = y_t - \hat{y}_t$, of all the samples. However, our problem also presents other challenges. The DVFS mechanism would benefit from anticipating workload changes [18]. And, as noted in Section 3.2, the performance of the workloads does not change frequently. Infrequent changes are a challenge because the training data indicates the model that it is correct to do same-value predictions most of the time. The ability to accurately predict changes is what would make any model stand out against same-value prediction.

We propose and explore different architectures of two dynamic performance forecasting models that use LSTM as their primary structure. Their names are Continuous Forecasting LSTM or CoFo, and Next Change LSTM or NxC. They both predict the value of CPI for one period ahead.

4.1 Continuous Forecasting LSTM

CoFo is a forecasting model that triggers the inference cycle of its LSTM at each DVFS period. It has an LSTM layer and one dense layer with linear activation. Its structure is very similar to the one shown in Figure 3.4, but the number of inputs and LSTM cells can be different.

The inputs of CoFo can be either one, a subset, or all the variables in our time series. While increasing the number of inputs increases the number of trainable parameters, they may provide valuable information to the LSTM forecasts if selected carefully. A correlation analysis is required to select inputs that do not introduce redundant information. When the input space is multi-

dimensional, principal component analysis (PCA) [2] is a way to reduce the dimensionality while keeping the essential information from the original data. Intending to minimize the number of trainable parameters as much as possible, CoFo explores using PCA when it has multiple inputs.

Another design choice that impacts the number of trainable parameters is the size of the network. The complexity of the functions that an ANN can represent depends on its size. Enabling more functions is a way of attempting to decrease the loss of the network. However, this can also result in over-fitting the training data. We increase the size of CoFo's LSTM by adding cell units; naturally, we select the number that produces the least average forecasting errors.

CoFo performs a small transformation to its inputs before feeding them to its neural network. It processes and forecasts relative values instead of absolute values. In other words, its input is $\Delta\hat{y}_t = y_t - y_{t-1}$ and it predicts $\Delta\hat{y}_{t+1}$. Thus, the LSTM learns to predict changes in CPI that CoFo converts back to absolute values.

4.2 Next Change LSTM

The observation that significant changes happen very infrequently in most of the workloads (as portrayed in Figure 3.1) motivated the design of NxC. By making all the samples in our traces available to an LSTM for training, it will most likely learn that predicting the same values is the best thing to do. NxC was designed to avoid feeding consecutive samples with very similar

values to the LSTM.

We propose grouping similar input values together and use run-length encoding (RLE) to feed them only once to the LSTM. Consequently, NxC's LSTM is not making predictions at each DVFS period. Instead, it is triggered only when a significant change in the input is detected, i.e., a sampled value belongs to a group different from the previous value.

Grouping requires a way of determining whether samples have similar values, which can be addressed in multiple ways. We could potentially adapt one of the phase classification mechanisms summarized in Chapter 2. However, for exploration purposes, we use buckets that represent different ranges of CPI values, all with the same size. Each bucket, b, holds a representative value, v_b, which is used to convert from the discrete to the continuous domain.

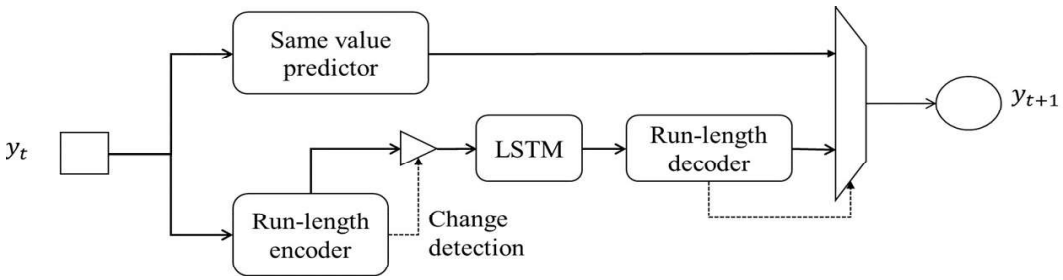

Figure 4.1: Algorithm for predicting with RLE and LSTM

Once the inputs are in discrete form, RLE can be applied, and use the encoded data as input to a predictor. Figure 4.1 shows the structure of NxC. The inference of the LSTM is started only when a change is detected. The RLE provides the LSTM with two values: a bucket, b_t, and a run-length, l_{t-1}.

Then, the LSTM predicts the duration of b_t, $\hat{l}_t$, and the next bucket $\hat{b}_{t+\hat{l}_t}$. A separate predictor can be used to forecast the values for $\hat{l}_t - 1$ periods. Then at time $t + \hat{l}_t$ the prediction $\hat{y}_{t+\hat{l}_t} = v_{\hat{b}_t}$ is the output.

As opposed to CoFo, NxC may process either scalar or one-hot encoded values. One-hot encoding is more commonly used in classification rather than regression problems. However, it has also been used for predicting sequences and finding patterns [13]. When the output is one-hot encoded, our ANN is solving a classification problem in which it returns the probabilities of the next sample, y_{t+1} belonging to each bucket. NxC selects the bucket with the highest probability and uses its value to return the forecast.

Chapter 5

Methodology

This chapter illustrates the process followed to obtain the forecasting models evaluated in Chapter 6. It includes the collection of training and validation data, the analysis of traces, and the tools used for architectural exploration.

We design our model considering that its predictions will be used by a software module. Intel processors provide autonomous management as a unit at the hardware level, called hardware-managed P-states (HWP), as well as interfaces for drivers or other software components to either instruct policies to HWP or disable it completely and make decisions on their own [15]. We consider sampling periods and data that are accessible at the software level of the computer architecture stack.

5.1 Data Collection

The time series was generated by collecting multiple event counters from the performance monitoring units (PMU) of a hardware platform. The target platform is an Intel's 10th generation processor running Ubuntu 18.04 operating system. Some of the architectural features are summarized in Table

5.1. Figure 5.1 exhibits the data collection process. It was repeated with different frequencies, sampling periods, and PMU events.

Table 5.1: Target hardware platform features

Threads per core	2	L1 data cache	48KB
Min frequency	400 MHz	L2 cache	512KB
Base frequency	1.3 GHz	LLC	2MB
Max. frequency	3GHz	Micro architecture	Sunny Cove

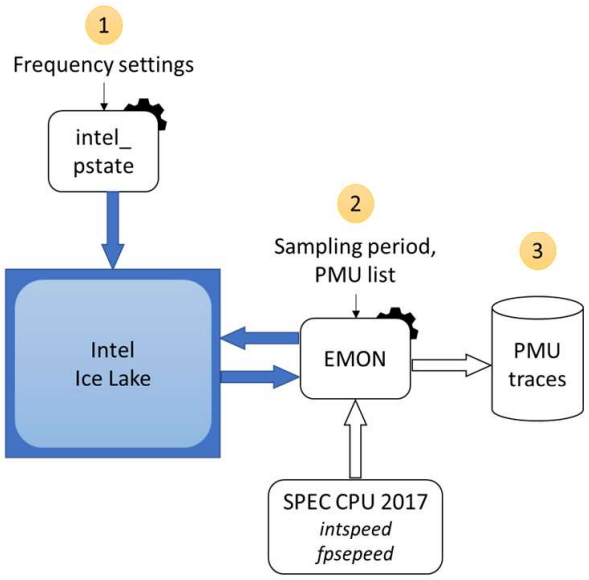

Figure 5.1: Data collection steps diagram. (1) Set frequency options with intel_pstate driver. (2) Select a sampling period and events list for EMON and run data collection for each benchmark. (3) EMON generates PMU traces.

5.1.1 Frequency Settings

The Linux kernel includes a driver called *intel_pstate* as part of the CPU performance scaling subsystem [28]. Its interface was used to configure

the frequency of each data set. A list of the different settings is shown in Table 5.2. The first column is the name that was given to each configuration for reference in this document.

All the listed configurations employ the *powersave* mode of *intel_pstate*. However, notice that when the minimum and maximum frequencies have the same value, the driver has no room for scaling, resulting in the core running at a constant frequency. The combination of *hwp_on* settings is equivalent to the *HWP+powersave* active mode, while *hwp_off* is equivalent to the *powersave* active mode from the *intel_pstate* documentation [28]. We did not change any other default configurations, including the sampling rate, whose default and minimum value is 10ms.

Table 5.2: List of different frequency configurations (GHz)

Name	HWP	Turbo	Min. freq	Max. freq
hwp_on	On	On	0.8	3.0
hwp_off	Off	On	0.8	3.0
0.8GHz	Off	Off	0.8	0.8
1.2GHz	Off	Off	1.2	1.2
1.5GHz	Off	Off	1.5	1.5
1.8GHz	Off	Off	1.8	1.8

5.1.2 Benchmarks

A subset of programs from the SPEC CPU® 2017 benchmark package [9] was used. For the scope of this project, the workloads are executed on one core because we want to generate a time series for an entire single run of

each. Consequently, the two SPECrate® suites were discarded and the subset of benchmarks was selected from both, Integer and Floating Point suites of SPECspeed®. An additional requirement was for the traces to be long enough to have a meaningful number of samples. Only traces that run for at least one minute are evaluated in this work. The subset includes *600.perlbench_s*, *605.mcf_s*, *641.leela_s*, and *657.xz_s* from *intspeed* and *607.cactuBSSN_s* and *644.nab_s* from *fpspeed*.

It is essential to disclose that SPEC's command-line tool *runcpu* was not utilized for running the benchmarks since it executes other modules that are not relevant to the performance traces. On the other hand, the reference workloads (input arguments) provided for each benchmark were used and not changed. Some benchmarks include multiple workloads, and if the index is not specified in the results shown in this document, only reference zero is used. The compiler version and flags used for each suite are shown in Table 5.3.

Table 5.3: Compiler flags (*icc, icpc, ifort*) version 18.0.2

Suite	**Flags**
intspeed	*-m64 -std=c11 -march=core-avx2 -ipo -O3 -no-prec-div -L$(JEMALLOC64_DIR) -l $(JEMALLOC64) -W,-z,muldefs -qopt-mem-layout-trans=3 -DSPEC_SUPPRESS_OPENMP*
fpspeed	*-m64 -std=c11 -march=core-avx2 -ipo -O3 -no-prec-div -qopt-prefetch -ffinite-math-only -qopt-mem-layout-trans=3 -DSPEC_SUPPRESS_OPENMP*

5.1.3 EMON

EMON [8] is a command-line tool developed by Intel which accesses and displays the contents of the PMU counters. EMON has a driver, which, if provided a time interval, accesses the counters periodically for as long as a given binary file is executing. The data sets analyzed in this project cover sampling periods starting at 1ms and up to 1s.

Depending on the hardware, there may be hundreds of PMU registers [16, 17]. However, each architecture constrains the number of events that can be concurrently accessed. In the case of Ice Lake, four fixed counters and up to 8 variable counters can be sampled per core (or uncore) at the same time. EMON takes a list of events as input and returns periodic samples of the counters for all the cores in the system (unless an uncore counter is specified in which case it is returned only once). A list of the performance counters that were collected in this project is displayed in Appendix A.

As mentioned earlier, each trace was generated from running a benchmark on a single core. To achieve this, we attached the execution of the binary to a known CPU using Linux's command-line *taskset*. Then, only the event counters of that core were extracted from EMON's output.

5.2 Feature Selection

We refer to the inputs of a model like features. The performance counters are transformed into features that are more robust to frequency changes.

Ideally, the features should not depend on the V/f selection. At the hardware level, previous work uses intervals of a fixed instruction count to collect the features, resulting in measurements that are less likely to be disturbed by frequency changes than those collected using time intervals. However, instruction count intervals are not accessible at the software level. Instead, we divide all the counter values in a sampling period by the instruction count of that period. We also explore the use of rates for the counters that is meaningful such as cache hit or miss rate. Some examples of this transformation are shown in Table 5.4. The features are defined in Equation 5.1. Similar transformations were performed to all the collected performance counters.

Table 5.4: Examples of PMU events to features transformation

PMU events names	**Features names**
CPU_CLK_UNHALTED.THREAD	CPI
BR_INST_RETIRED.ALL_BRANCHES, BR_MISP_RETIRED.ALL_BRANCHES	BR_PI, BR_MPI, BR_MR

$$
\begin{aligned}
CPI &= \frac{CPU_CLK_UNHALTED.THREAD}{INST_RETIRED.ANY} \\
BR_PI &= \frac{BR_INST_RETIRED.ALL_BRANCHES}{INST_RETIRED.ANY} \\
BR_MPI &= \frac{BR_MISP_RETIRED.ALL_BRANCHES}{INST_RETIRED.ANY} \\
BR_MR &= \frac{BR_MISP_RETIRED.ALL_BRANCHES}{BR_INST_RETIRED.ALL_BRANCHES}
\end{aligned}
\tag{5.1}
$$

The number of input features that can be used for a model is limited by the number of performance counters that can be sampled simultaneously.

Thus, it is essential to select features that are not redundant. Multiple sets of counters were generated, and for each set, a correlation table was used to discard highly correlated features. The steps followed in the process of feature selection are:

1. Select, arbitrarily, a set of PMU events that can be sampled simultaneously.

2. Collect the traces of all benchmarks following the process from Figure 5.1 and get the features of each trace.

3. Calculate the absolute value of the Spearman's correlation coefficient of each pair of features for every trace. The absolute value is used because we care about the magnitude of the correlation only.

4. Create a table of the coefficients of each pair of features with their average across all benchmarks. An example of the output of this step is shown in Figure 5.2.

5. If there are pairs that have an average correlation greater than 0.70, discard one of them. If discarding a feature results in a nonutilized counter, arbitrarily select a new PMU event to replace it and go back to step 2.

As an example, examine the features shown in Figure 5.2. Each value depicts an average across all benchmarks. The color scheme shows darker

	CPI	L3_MPI	BR_PI	BR_MPI	UOPS_PI	UOPS_SPI	BR_MR
CPI	1.00	0.49	0.47	0.70	0.56	0.85	0.62
L3_MPI	0.49	1.00	0.34	0.32	0.30	0.55	0.38
BR_PI	0.47	0.34	1.00	0.53	0.42	0.39	0.56
BR_MPI	0.70	0.32	0.53	1.00	0.61	0.51	0.88
UOPS_PI	0.56	0.30	0.42	0.61	1.00	0.39	0.52
UOPS_SPI	0.85	0.55	0.39	0.51	0.39	1.00	0.42
BR_MR	0.62	0.38	0.56	0.88	0.52	0.42	1.00

Figure 5.2: Example of a correlation table of a subset of features

shading for higher values. Notice that the table is diagonally mirrored. The values highlighted in white font on the left side of the diagonal are high correlation values. Four features are candidates to discard: CPI, BR_MPI, UOPS_SPI, and BR_MR. CPI is formed by two of the four fixed counters in Ice Lake, hence removing this feature does not result in gaining a new space to collect new events. It is preferred to discard the features highlighted with red fonts. Dropping UOPS_SPI results in a nonutilized event for this trace, UOPS_ISSUED.STALL_CYCLES; however, BR_MPI shares a counter with BR_MR, thus discarding BR_MPI does not result in nonutilized counters. For this example, one new PMU event would be arbitrarily selected to replace UOPS_ISSUED.STALL_CYCLES and steps 2 - 5 would then be repeated.

Finally, a primary variable of interest is required, i.e., a feature to forecast. In this project, CPI is selected because it is one of the metrics that has been used in previous work to represent program phases. However,

we acknowledge that a more in-depth study, out of the scope of this work, may be required to understand the metric that works best with a particular DVFS controller. Unless otherwise specified, the rest of this document refers to forecasts of CPI values.

5.3 Metrics and Baselines

This section describes the metrics used for this project: root mean squared error (RMSE), change confusion matrix, and complexity.

5.3.1 Root Mean Squared Error

RMSE is a commonly used metric for regression and forecasting of continuous values [23]. It has the advantage of giving a single numeric value to compare across multiple models. By representing the average magnitude of the error, RMSE is a good indicator of how predictors compare in the average case. It is defined as:

$$RMSE = \sqrt{\frac{1}{N}\sum_{t=1}^{T}(y_t - \hat{y}_t)^2} \tag{5.2}$$

Note that the limits of the summation can be different depending on whether there is a region of interest in the time series.

5.3.2 Change Confusion Matrix

It was motivated in chapter 3 that the main challenge of our models is to predict changes. Thus, we need a metric to evaluate this challenge. We present the change confusion matrix. It has the same purpose as a confusion matrix that is used to evaluate classification models. We use three classes: *up*, *down*, and *no_change*. They are similar to the definition of *change* introduced in Section 3.2. The difference is that the *change* class is divided into two, *up* or positive and *down* or negative change.

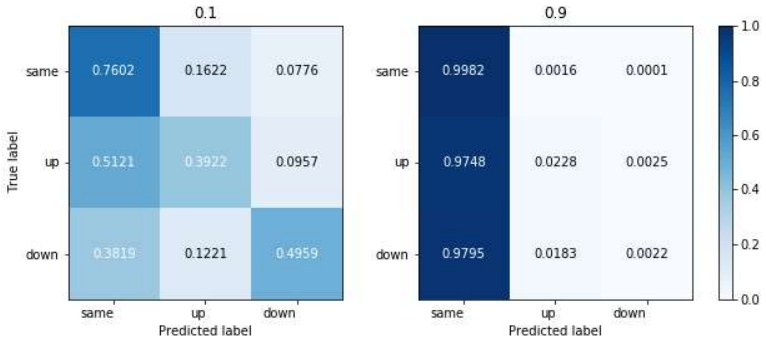

Figure 5.3: Confusion matrices of two EWMA predictors: $\alpha = 0.1$ (left) and $\alpha = 0.9$ (right)

Two confusion matrices are displayed in Figure 5.3. Each row contains the fraction of times a class was predicted (x-axis) when the true class (y-axis) was the class of that row. The sum of the three values on each row is 1. Both matrices correspond to forecasts made by EWMA with parameters of $\alpha = 0.1$ and $\alpha = 0.9$. They were generated by averaging the confusion matrices of all

benchmarks.

5.3.3 Complexity

The models used for time series forecasting applications are very powerful, but they can also be very expensive in terms of computation and memory usage. Because the consumer of our predictor is a DVFS policy whose goals are usually to either minimize power or maximize performance, we do not want to introduce overheads that will harm either of them. Thus, keeping the model complexity as low as possible is one of our design constraints.

In our models, we refer to complexity as the number of trainable parameters because it represents a reasonable estimate of the cost of the model. Having more trainable parameters translates into storing more data as well as performing more operations. The complexity of EWMA, for example, is 1 because it has one trainable parameter, α.

5.4 Frequency and Sampling Rate Selection

DVFS is the target application of this work, which means the forecasting model should be robust to frequency changes. In this section, we motivate the challenge for a predictor when scaling frequency. Additionally, we also briefly explore the potential challenges of sampling rates other than *intel_pstate*'s default, 10ms.

We studied the RMSE of same-value prediction with the frequency settings introduced in Table 5.2 in Figure 5.4. For most of the benchmarks,

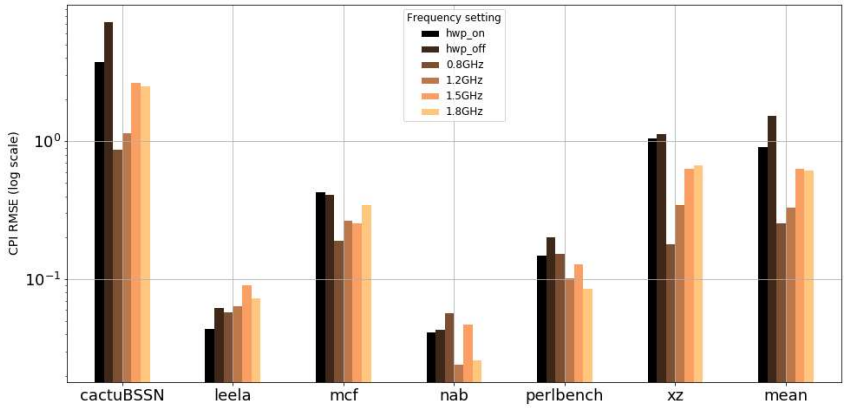

Figure 5.4: RMSE of CPI same value predictor for different frequency settings

the error of traces generated with a fixed frequency is significantly lower than traces with frequency scaling enabled. We find opportunity in the decrease of same-value prediction accuracy when the frequency is changing, and thus we evaluate our models using those traces.

The sensitivity of CPI same-value prediction to different sampling periods is shown in Figure 5.5. While looking at the figure, it is hard to find a consistent relationship between the magnitude of the prediction error and the sampling period. If we refer back to Figure 3.1, the RMSE in predictions is higher when the fraction of observed changes increases, which is expected for a same-value predictor.

The changes that may occur in the workload are always filtered by using coarse sampling periods. The purpose of collecting data at a granular-

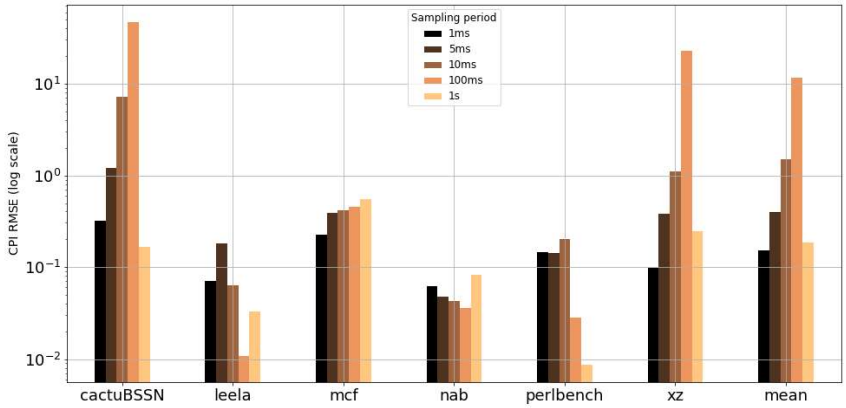

Figure 5.5: RMSE of CPI same value predictor for different sampling periods

ity smaller than 10ms was to find whether more changes could be observed; however, the opposite happened. The possibility of these changes existing at higher rates is not discarded, but increasing the sampling rate at the software level may result in undesired overheads. Thus, we limit the scope of this work to use the same sampling period that *intel_pstate* provides. We believe that since the error of 100ms is higher than 10ms, the forecasting model may exhibit more significant accuracy improvements. However, in this project, we use the default 10ms for evaluation purposes.

5.5 Training and Validation Framework

The following list briefly describes our machine learning framework:

- We use the *python* library *keras* [6] running with *tensorflow* [1] on the

back end.

- Each trace is trained and validated separately from the rest.
- We use the 70/30 rule for splitting traces into training and testing data.
- The features of our models are scaled to values between $[-1, 1]$ when using scalar inputs for the LSTM because the outputs of its internal activation functions are values in this range.

Chapter 6

Evaluation

This chapter presents the evaluation of EWMA, CoFo, and NxC with respect to same-value prediction. It also shows a few essential exploration steps that lead to the best configurations of the LSTM.

6.1 Exponentially Weighted Moving Average

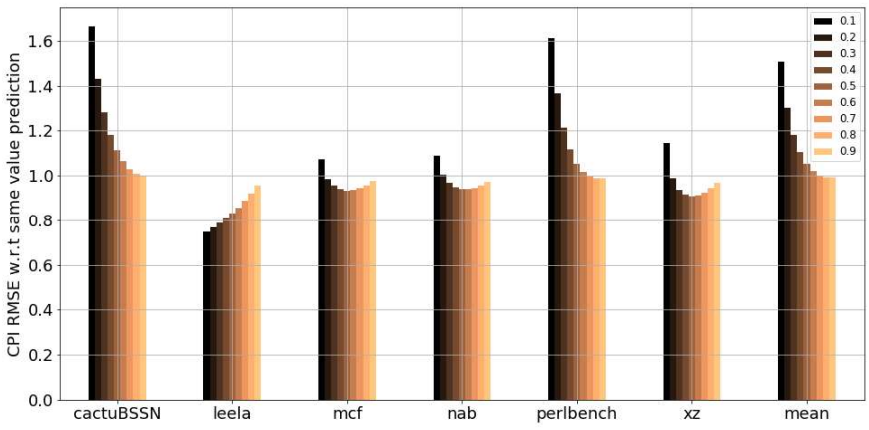

Figure 6.1: Sensitivity study of α for forecasting with EWMA

Figure 6.1 compares the RMSE of EWMA and same value prediction. It shows the RMSE of different values of α normalized to the RMSE of same-

value prediction. The right-most columns of Figure 6.1, labeled as "mean", show the mean across all benchmarks.

The recursive form of EWMA, introduced in Eq. 3.3, was implemented. It has two components, one represents the history, and the other is the current observation. A greater value of α gives higher weight to the current observation. The best configuration is $\alpha = 0.9$, with an error decrease of 3%.

In Section 5.3, we showed the confusion matrices of two configurations of EWMA. Notice that, even though $\alpha = 0.1$ can predict more changes, Figure 6.1 shows that it is not a better predictor in the average case. Therefore, we learn that sacrificing the accuracy of same-value prediction for being able to predict changes may not result in a better predictor.

6.2 Continuous Forecasting LSTM

This section evaluates two necessary exploration steps for CoFo: input selection and network size. Both design decisions impact the count of trainable parameters and may introduce a trade-off between complexity and forecast error.

6.2.1 Input Selection

CoFo only uses scalar inputs and outputs. The different inputs explored for CoFo are three: **cpi**, **spearman**, and **all**. The first one uses CPI as its only input. The second one uses the features that have a Spearman correlation lesser than 0.5. And, as its name indicates, **all** uses all the features of the trace.

For **spearman** and **all**, we also studied the effects of applying PCA with a variance of 90%. The results are shown in Figure 6.2. They are compared to the best EWMA configuration.

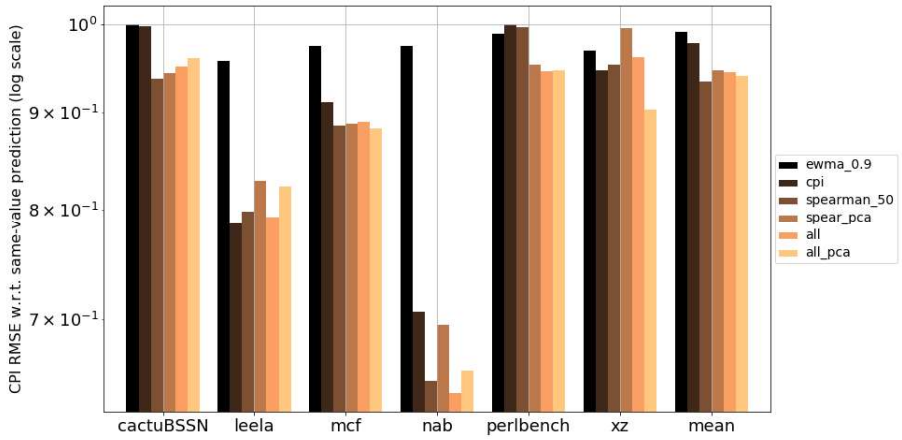

Figure 6.2: Study of the impact of input selection and PCA on forecasting error of CoFo. The y-axis is the RMSE with respect to same-value prediction

Adding input features results in lowering the forecasting error. The experiment that uses a single input, **cpi**, has the least average error decrease of 2%, while the best one, with 5 inputs, decreases the error by 7%. The value of using LSTM is more noticeable for *nab* where the same-value prediction RMSE is reduced by 35%. This benchmark has a pattern that is repeated throughout most of the traces, proving the success of CoFo in recognizing patterns. *cactuBSSN* also has a pattern that is repeating, but it is repeated roughly every 2000 samples as opposed to *nab*'s repetition length of roughly

500. This was a motivation to use RLE for NxC.

To understand what is being predicted in each model, we look at the confusion matrices in Figure 6.3. We notice that **cpi** has the highest rate of *same* prediction, whereas using other features seem to be giving information about when to predict changes. On the other hand, using more input features results in increasing complexity. Table 6.1 shows the number of trainable parameters for the different experiments.

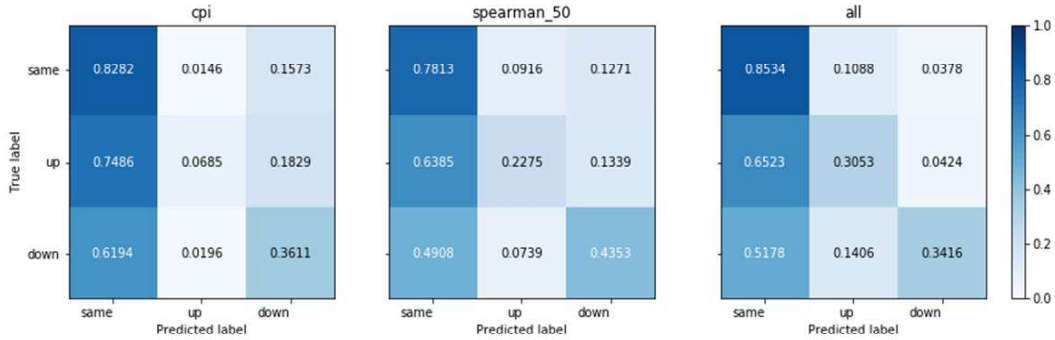

Figure 6.3: Confusion matrices of different inputs to CoFo LSTM

Table 6.1: Complexity of single cell experiments

Experiment name	Complexity
cpi	14
spearman_50	30
spear_50pca	22
all	54
all_pca	26

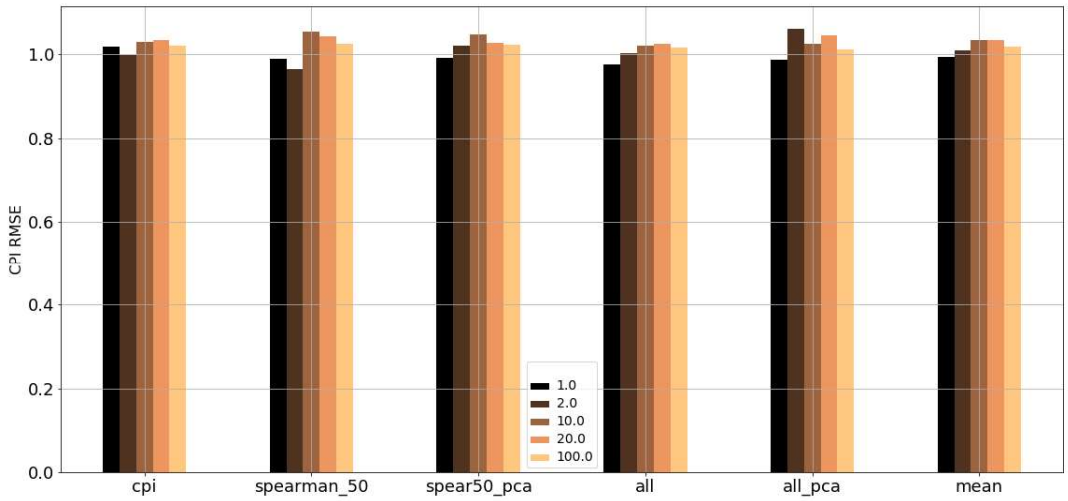

Figure 6.4: Comparison of forecasting error of CoFo with multiple LSTM cells

6.2.2 Network Size

Increasing the complexity by using more inputs resulted in lower error estimation. Another way to increase complexity is by using more LSTM cells. We found that, conveniently, the error decreases while the network size decreases, too. The results of adding extra LSTM cells to CoFo are shown in Figure 6.4.

Because the error is only reduced when adding more features, we conclude that CoFo can be improved by exploring more PMU events and their features, but we leave this for future work. Some results of changing training parameters that do not impact network size are exhibited in Appendix B.

6.3 Next Change LSTM

The design of NxC has multiple variables. Surprisingly, selecting the number of buckets did not result in significant changes in the RMSE. On the other hand, the representation of the inputs and outputs to the network, and whether or not to apply differencing impacted the RMSE significantly. The inputs and outputs are given in either scalar (sc) or one-hot (oh) format. The number of trainable parameters of each configuration is shown in Table 6.2.

Table 6.2: Complexity of NxC with different input and output encodings

Input	Output	Complexity
scalar	scalar	28
scalar	one-hot	62
one-hot	one-hot	130

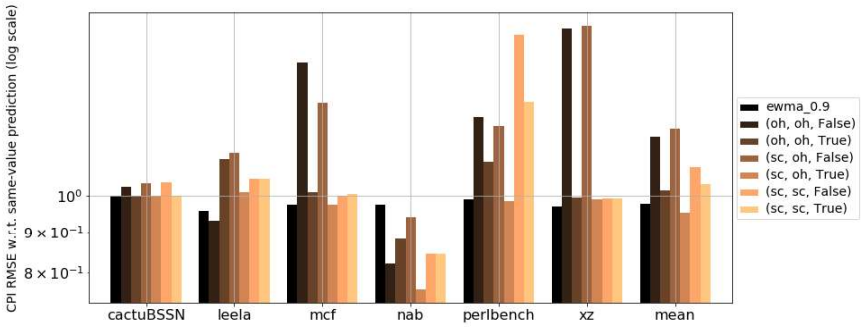

Figure 6.5: Study of the impact of different input configuration on forecasting error of NxC compared to EWMA. The y-axis is the RMSE with respect to same-value prediction

Figure 6.5 shows the RMSE relative to same-value prediction. The

number of buckets used is 16. The labels for NxC configurations have the form (i, o, d) where i and o are the input and output encodings, respectively, and d is the boolean for whether differencing was applied to the data. The mean across all benchmarks of the best configuration (sc, oh, False) shows that error was decreased by 5% with respect to same-value prediction, as opposed to 1% of EWMA.

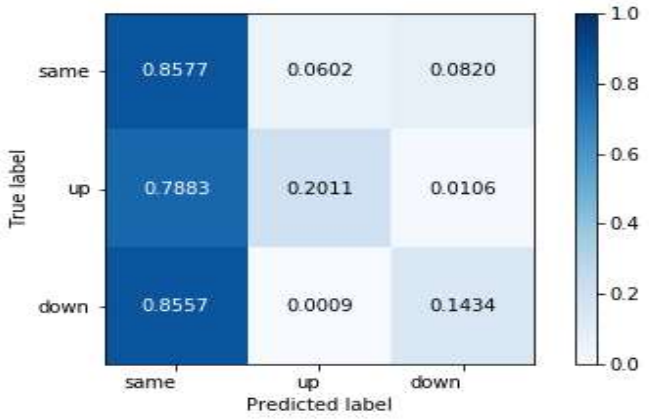

Figure 6.6: Change confusion matrix of NxC

When we look at the confusion matrix in Figure 6.6, NxC can predict more changes than EWMA, but is not predicting as many as CoFo. When we took a closer look, we realized that the LSTM wrapped inside NxC is predicting only one bucket most of the time because RLE is not robust to outliers. When a CPI level is interrupted for a period, the RLE encoder classifies this as a change. Then, it feeds three values to the LSTM, out of which two are the same. If this happens multiple times throughout the trace, there is one bucket

that the LSTM sees as input most of the time, and it learns to predict that bucket most of the time. For future work with LSTM+RLE, we expect to find a way to detect disturbances so that RLE gives more robust inputs to the LSTM.

CoFo gives better results than NxC. Its best configuration reduces the RMSE by 7% on average, as opposed to NxC's 5%. Conveniently, the complexity of CoFo is lower than NxC's, which allows us to provide higher accuracy with lower complexity.

Chapter 7

Conclusion and Future Work

This project sets the goal of accurately forecasting the dynamic performance of a workload periodically at running-time, keeping in mind that a DVFS module uses the forecasts. We evaluated three models and compared them to same-value prediction. All of them decreased the forecast error, and the best one, which is based on LSTM, decreased the RMSE by 7% on average.

We motivated the challenge of dynamic performance forecasting as being able to predict workload changes. Our models were able to predict some of them, but there is still room of opportunity for future work.

In this forecasting problem, increasing the number of training parameters of an LSTM did not always result in decreasing RMSE. Thus, we present a low-complexity LSTM-based model for periodically predicting CPI values. The case where increasing complexity resulted in better results is when we used more features as inputs to the network. Thus, we believe that in future work, studying more features and filtering the ones whose information is more valuable may result in better results.

We also consider, for future work, that the outcomes of this project can be adapted and applied to other applications such as performance modeling.

Appendices

Appendix A

List of collected PMU counters

BR_INST_RETIRED.ALL_BRANCHES

BR_MISP_RETIRED.ALL_BRANCHES

CPU_CLK_UNHALTED.THREAD

DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK

FP_ARITH_INST_RETIRED.SCALAR_DOUBLE

FREERUN_PKG_ENERGY_STATUS

INST_RETIRED.ANY

L2_RQSTS.DEMAND_DATA_RD_MISS

L2_RQSTS.ALL_DEMAND_DATA_RD

L2_RQSTS.DEMAND_DATA_RD_HIT

LONGEST_LAT_CACHE.MISS

OFFCORE_REQUESTS.L3_MISS_DEMAND_DATA_RD

UOPS_EXECUTED.CORE

UOPS_ISSUED.STALL_CYCLES

Appendix B

Hyper-parameter selection study

Below we present the results of changing three hyper-parameters for CoFo: loss function, optimizer, and batch size

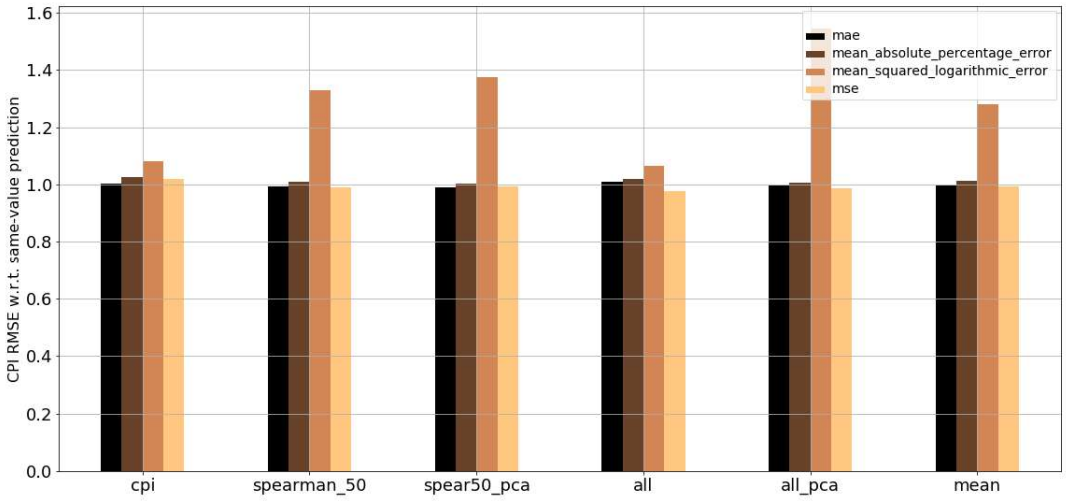

Figure B.1: RMSE of LSTM predictors with a single cell and use of different loss functions

Keras offers four pre-defined loss functions for regression problems [6]. The results after training a model with each of them are shown in Figure B.1. The winner is mean squared error (mse), followed by mean absolute error (mae). The graph shows that the worst choice is mean squared logarithmic

error, because we use scaled values for training.

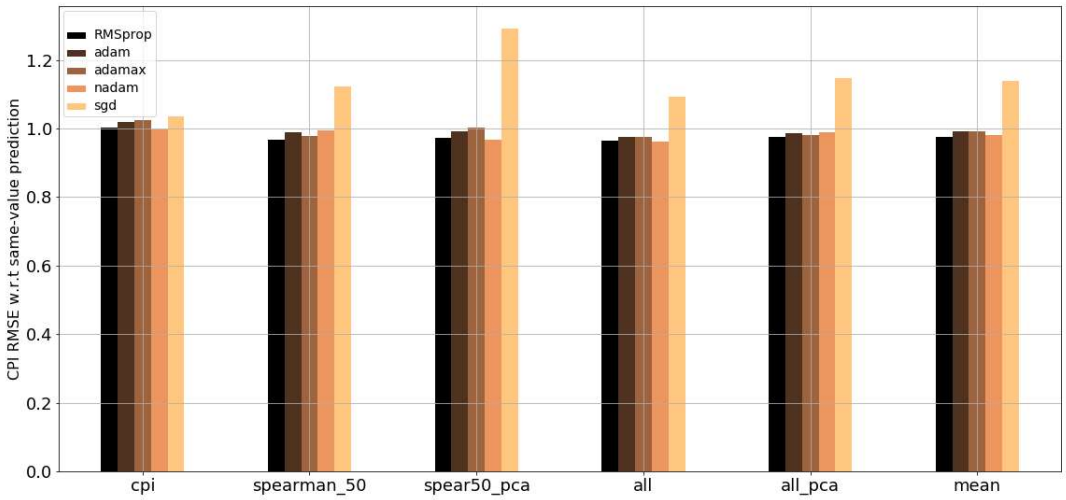

Figure B.2: RMSE of LSTM predictors with a single cell and different optimizers

We also chose four optimizers provided by keras and trained our models with their design parameters. The RMSE of CPI forecasting with respect to same-value prediction is shown in Figure B.2. The winner is RMSprop. And the poorest choice is stochastic gradient descent.

Finally, changing the training batch size also made a difference in our results. We arbitrarily explore four batch sizes between 1 and 3000. The results are shown in Figure B.3. Notably, the RMSE error increases with the batch size.

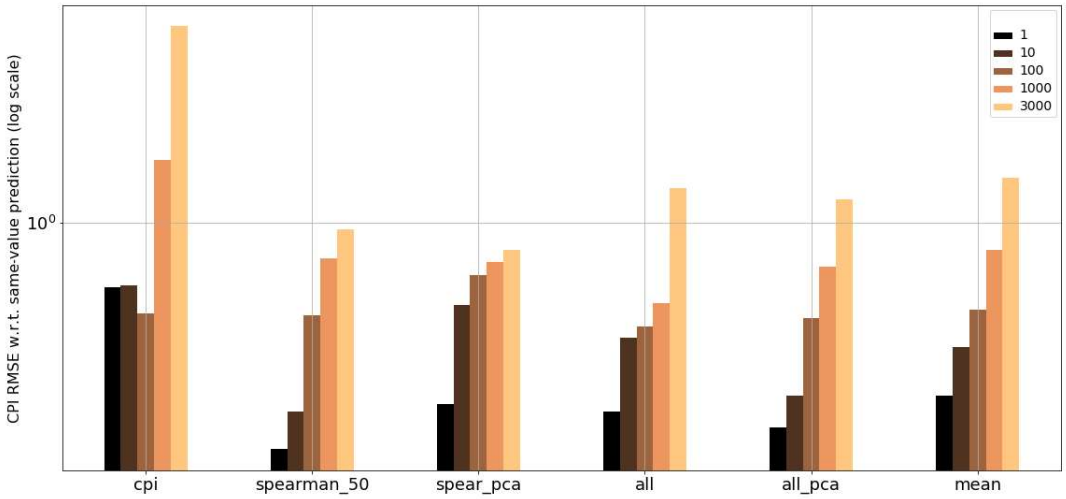

Figure B.3: RMSE of LSTM predictors with a single cell and different training batch sizes

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Hervé Abdi and Lynne J. Williams. Principal component analysis. *WIREs Comput. Stat.*, 2(4):433–459, July 2010.

[3] I. Anghel, T. Cioara, I. Salomie, G. Copil, D. Moldovan, and C. Pop. Dynamic frequency scaling algorithms for improving the cpu's energy efficiency. In *IEEE 7th International Conference on Intelligent Computer Communication and Processing (ICCP)*, pages 485–491, August 2011.

[4] C. Chang, P. Liu, and J. Wu. Sampling-based phase classification and prediction for multi-threaded program execution on multi-core architec-

tures. In *42nd International Conference on Parallel Processing*, pages 349–358, October 2013.

[5] Yen-Lin Chen, Ming-Feng Chang, Chao-Wei Yu, Xiu-Zhi Chen, and Wen-Yew Liang. Learning-directed dynamic voltage and frequency scaling scheme with adjustable performance for single-core and multi-core embedded and mobile systems. *Sensors*, 18:3068, September 2018.

[6] François Chollet et al. Keras. `https://keras.io`, 2015.

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

[8] Intel Corporation. Emon user guide. `https://software.intel.com/en-us/download/emon-user-guide`, November 2019.

[9] Standard Performance Evaluation Corporation. Spec cpu®2017. `https://www.spec.org/cpu2017/index.html`, September 2019.

[10] M. T. Cruz, P. Tomás, and N. Roma. Unsupervised variable-grained online phase clustering for heterogeneous/morphable processors. In *International Conference on High Performance Computing Simulation (HPCS)*, pages 858–865, July 2016.

[11] Gaurav Dhiman and Tajana Simunic Rosing. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In *International*

Symposium on Low Power Electronics and Design (ISLPED), pages 207–212, New York, NY, USA, 2007. ACM.

[12] John Cristian Borges Gamboa. Deep learning for time-series analysis. *Computing Research Repository*, abs/1701.01887, 2017.

[13] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. *Computing Research Repository*, 2018.

[14] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[15] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2.* Number 253669-060US. September 2016.

[16] Intel Corporation. *Intel® 64 and IA32 Architectures Performance Monitoring Events.* Number 335279-001. December 2017.

[17] Intel Corporation. *Intel® Xeon® Processor Scalable Memory Family Uncore Performance Monitoring Reference Manual.* Number 336274-001US. July 2017.

[18] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 359–370, December 2006.

[19] R. Jain, P. R. Panda, and S. Subramoney. Machine learned machines: Adaptive co-optimization of caches, cores, and on-chip network. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 253–256, March 2016.

[20] Da-Cheng Juan and Diana Marculescu. Power-aware performance increase via core/uncore reinforcement control for chip-multiprocessors. In *ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, pages 97–102, New York, NY, USA, 2012.

[21] Kai Ma and Xiaorui Wang. Pgcapping: Exploiting power gating for power capping and core lifetime balancing in cmps. In *21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 13–22, September 2012.

[22] Omer Khan and Sandip Kundu. *Microvisor: A Runtime Architecture for Thermal Management in Chip Multiprocessors*, pages 84–110. Springer Berlin Heidelberg, 2011.

[23] Douglas C. Montgomery. *Introduction to Time Series Analysis and Forecasting (Wiley Series in Probability and Statistics)*. Wiley-Interscience, April 2015.

[24] D. M. Q. Nelson, A. C. M. Pereira, and R. A. de Oliveira. Stock market's price movement prediction with lstm neural networks. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1419–1426, May 2017.

[25] Nong Ye, Qiang Chen, and C. M. Borror. Ewma forecast of normal system activity for computer intrusion detection. *IEEE Transactions on Reliability*, 53(4):557–566, December 2004.

[26] S. Selvin, R. Vinayakumar, E. A. Gopalakrishnan, V. K. Menon, and K. P. Soman. Stock price prediction using lstm, rnn and cnn-sliding window model. In *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 1643–1647, September 2017.

[27] S. Srinivasan, R. Kumar, and S. Kundu. Program phase duration prediction and its application to fine-grain power management. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 127–132, Aug 2013.

[28] Rafael J. Wysocki. intel_pstate cpu performance scaling driver. `https://www.kernel.org/doc/html/v4.12/admin-guide/pm/intel_pstate.html`, 2017.

[29] Z. Zhao, W. Chen, X. Wu, P. C. Y. Chen, and J. Liu. Lstm network: a deep learning approach for short-term traffic forecast. *IET Intelligent Transport Systems*, 11(2):68–75, 2017.

Index

Abstract, vi
Acknowledgments, v
Appendices, 44

Benchmarks, 23
Bibliography, 53

Change confusion matrix, 30
CoFo, 17

Data collection, 21
Dedication, iv
DVFS, 4

EMON, 25
EWMA, 10

Forecasting models, 8
Forecasting notation, 8

intel_pstate, 22
Introduction, 1

Methodology, 21

NxC, 18

Phase prediction, 5

Related work, 4
RMSE, 29

Same value prediction, 9

Vita

Erika Susana Alcorta received her Bachelor of Science degree in Digital Systems and Robotics Engineering from the Monterrey Institute of Technology and Higher Education (Tec de Monterrey). She started her career designing embedded systems for a radio frequency identification company. Then, she explored a career in software development by working as a Virtual Platforms Software Engineer. In 2016, she applied to the Fulbright-Garcia-Robles scholarship for graduate school programs in the USA and received the grant in June of that year. She is currently enrolled as a Ph.D. student at the University of Texas at Austin. Her research interests include the application of machine learning to modeling and computer architecture problems.

Permanent address: esalcort@utexas.edu

This report was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.