MACHINE LEARNING TECHNIQUES FOR PERFORMANCE PREDICTION AND DIAGNOSIS OF VLSI DESIGNS

A Dissertation

by

ERICK MAURICIO CARVAJAL BARBOZA

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Jiang Hu
Co-Chair of Committee,	Paul Gratz
Committee Members,	Ulisses Braga Neto
	Duncan Walker
Head of Department,	Miroslav Begovic

December 2021

Major Subject: Computer Engineering

ABSTRACT

As the cost of scaling-down the manufacturing process of integrated circuits grows larger and its performance gains become smaller, designs must grow in complexity in order to achieve expected performance improvements. As this complexity grows, the development of automation tools for design, validation, and debug is critical. The number of machine learning-based techniques aiming to improve available tools has grown rapidly in recent years, as machine learning has proven an extraordinary capability of extracting knowledge from data and handling complicated non-linear behaviors, which makes it the best approach to mimic a human manual process among mathematical or algorithmic options. The work presented in this dissertation aims to evaluate the application of machine learning techniques in two different areas of the integrated circuit design process: pre-routing timing prediction and performance debugging of microprocessor cores.

The strategy proposed for pre-route timing prediction is based on machine learning models that predict the post-routing timing using only placed, but un-routed circuit databases. This strategy prevents over-design due to pessimistic timing estimations, as well as it saves time by reducing the need of multiple design iterations caused by the use of inaccurate timing estimations to guide circuit optimizations such as gate resizing, logic restructuring, or threshold voltage assignment leading to design violations once routing is executed. The obtained results show that our models achieve a prediction quality on-par with a sign-off static timing analysis commercial tool, with a $3\times$ speedup.

For the performance debug of microprocessor cores task, we focus on bugs that affect the generation-by-generation performance improvement in new designs. This task is very challenging due to the lack of an accurate golden performance model, unlike its functional counterpart. In addition, there is a limited visibility of the performance on intermediate steps of the design, and overall, the debugging infrastructure is lacking, which makes this problem even more challenging. Currently this process is executed on a highly manual manner, which requires large amounts of time to fully characterize a bug. Therefore, automated techniques for performance debugging are

essential to keep-up the performance gains obtained by new microarchitectural designs. In this dissertation, we focus on detecting the presence of a performance bug and localizing the microarchitectural unit on which the bug might be, more detailed debugging is left for future work. Our proposed techniques achieved up to a 91.5% of bug detection, and up to a 98% top-3 (out of 16 possible) bug localization accuracy on bugs with average IPC impact $> 1\%$.

DEDICATION

To Tía Maru and Abuelo Victor, up in heaven.

To my mother, my father, and especially to my wife.

ACKNOWLEDGMENTS

I would like to thank Dr. Jiang Hu, not only for his academic advice and support, but also for believing in me during trying times and for motivating me to push forward, without him this thesis would have never happened. I would also like to thank Dr. Paul Gratz, whose guidance to navigate the world of Computer Architecture has been essential.

I am also very thankful to Dr. Mahesh Ketkar and Dr. Michael Kishinevsky from Intel Labs. Not only were they amazing mentors throughout my research at Texas A&M, but they also provided me with the amazing opportunity to intern with them. Even with all the complications due to COVID-19, they were always eager to help.

I would also like to thank my committee members, Dr Braga Neto and Dr Walker for their helpful suggestions and comments.

Finally, I would also like to thank the University of Costa Rica for giving me the opportunity to do my graduate studies abroad, and specially Dr. Lochi Yu, for guiding me towards my academic path.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Jiang Hu and Professor Paul Gratz of the Department of Electrical and Computer Engineering, who served as co-advisors, and Professor Ulisses Braga Neto of the Department of Electrical and Computer Engineering and Professor Duncan Walker of the Department of Computer Science and Engineering.

The work in Chapter 2 was advised by Dr. Jiang Hu of Texas A&M University. Some test cases for this chapter were prepared by Nischal Shukla of Texas A&M University. This work was published in [1].

The work in Chapter 3 was advised by Dr. Jiang Hu, and Dr. Paul Gratz of Texas A&M University, and Dr. Mahesh Ketkar, and Dr. Michael Kishinevsky of Intel Labs. The work on performance bug detection shown in that chapter was published in [2].

Funding Sources

Graduate study was partially supported by Semiconductor Research Corporation (SRC) Tasks 2810.021 and 2810.022 through UT Dallas' Texas Analog Center of Excellence (TxACE), Semiconductor Research Corporation (SRC) Task 2902.001, and the University of Costa Rica, *"Graduate Studies Abroad for UCR Faculty"* Scholarship.

NOMENCLATURE

AMAT	Average Memory Access Time
AUC	Area Under the Curve
CBC	Counter-Based Classification
CNN	Convolutional Neural Network
CPI	Cycles Per Instruction
CTS	Clock Tree Synthesis
FF	Flip-Flops
FNR	False-Negative Rate
FPR	False-Positive Rate
GAN	Generative Adversarial Network
GBT	Gradient Boosted Trees
HDL	Harfware Description Language
IC	Integrated Circuits
IPC	Instructions committed Per Cycle
ITC	International Test Conference
LSTM	Long Short-Term Memory
MAE	Mean Absolute Error
ML	Machine Learning
MLP	Multi-Layer Perceptron
MSE	Mean Squared Error
P2BC	Performance Prediction error-Based Classification
PERT	Program Evaluation and Review Technique

RC	Resistor-Capacitor
RF	Random Forest
RMAE	Relative Mean Absolute Error
RMSE	Relative Mean Squared Error
ROC	Receiver Operating Characteristic
RRMSE	Relative Root Mean Squared Error
STA	Static Timing Analysis
SVM	Support Vector Machine
TNR	True-Negative Rage
TNS	Total Negative Slack
TPR	True-Positive Rate
VHDL	Very High-Speed integrated circuit Hardware Description Language
VLSI	Very-Large Scale Integration

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION AND MOTIVATION

With the ever-growing complexity of integrated circuits (ICs), the development of efficient automation tools in areas such as design, validation, and debugging is vital. In recent years, the number of machine learning (ML)-based techniques aiming to improve state-of-the-art tools has grown at a fast pace and has proven its value across a wide variety of problem formulations in almost every step of the IC design flow [3]. This dissertation proposes and evaluates the application of machine learning-based techniques that aim to improve two design aspects, specifically timing prediction for IC designs using pre-routing information, and performance debugging of microprocessor cores at microarchitectural level.

1.1 Pre-routing timing prediction

During the cell placement stage of the IC design flow, several optimizations, such as logic restructuring, gate resizing, and threshold voltage assignment, need to be executed in order to achieve the required timing and power specifications. All these optimization procedures require accurate estimations of the circuit timing, however, since routing information is essential for these, it is very challenging to achieve a satisfactory solution at early design stages.

To overcome this issue, designers typically employ one of two options:

1. To be overly-cautious, and use pessimistic estimations of the timing, this is to ensure that no violations will occur once routing is performed. An example of a commercial tool slack estimation on pre-routing databases is shown in Figure 1.1. The figure corresponds to a circuit with about 25k nets and 500 Flip-Flops (FFs). In the figure, the x-axis represents the worst path slack, as calculated by a post-routing sign-off tool, and the y-axis shows the worst path slack, as calculated by a commercial tool using a placed, but un-routed database. Each dot in the figure represents a different FF. Therefore, the red line indicates a perfect match between pre-routing estimation and post-routing sign-off analysis. In this example, the worst-case slack is over-estimated by more than 2ns. Such pessimism causes over-design

for area and power and wastes optimization time.

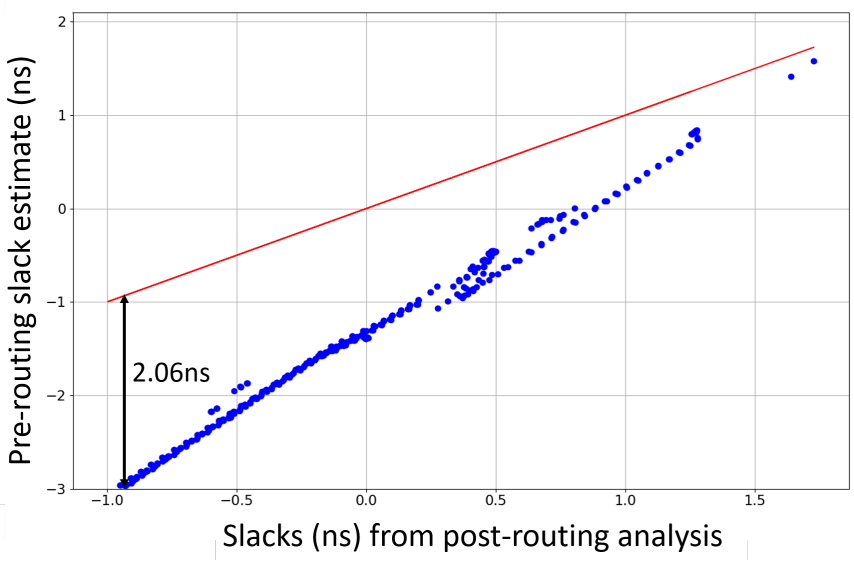


Figure 1.1: Pessimism of a commercial tool in pre-routing timing estimation. Reprinted from [1] © 2019 IEEE.

2. To run the whole flow, and iterate back to fix cell placement if the specifications were not met at routing. In this case, each iteration required will significantly increase the design turn-around time, and there is no guarantee that design convergence will be achieved after a certain number of iterations.

Taking all this into consideration, the goal of our research is to develop a machine learning model, that is able to perform accurate and fast predictions of the post-routing timing using placed, but un-routed circuit databases [1]. This will aid designers by reducing the design turn-around time, without incurring in over-design penalties.

1.2 Performance debugging of microprocessor cores

During the design of a new microprocessor, the process of verification and validation are typically the ones consuming most of the engineer's time and effort. These procedures can be divided

into two very distinct disciplines, namely, functional and performance verification. Functional verification handles design correctness and has received significant attention in the literature. Although challenging, it has the advantage of having known correct outputs to compare against. On the other hand, performance verification, which handles generation-by-generation workload performance improvement, lacks an exact golden reference for comparison. This is because it remains very difficult to model the interactions among different units in the highly complex microprocessor designs and accurately represent how they affect the system performance. The process of debugging also suffers from the very limited visibility on intermediate points, exposed mostly by performance counters, and the lack of a good debugging infrastructure. Currently, performance verification is conducted mostly through manual techniques that rely on rough estimates of performance gain expected by microarchitectural changes [4]. Such manual processes are not only very lengthy but also error prone.

Although simulation studies may provide an approximation of how long a given application might run on a real system, prior work has shown that these performance estimates will typically diverge from real systems performance [5, 6, 7]. Thus, the process to detect and analyze performance bugs is, in practice, via tedious and complex manual analysis, often taking large periods of time to isolate and eliminate a single bug. For example, consider the case of the Xeon Platinum 8160 processor snoop filter eviction bug [8]. Here the performance regression was more than 10% on several benchmarks. The bug took several months of debugging effort before being fully characterized. In a competitive environment where time to market and performance are essential, there is a critical need for more automated mechanisms that help designers to do performance debugging.

As an early work in this space, we have limited the focus of this dissertation mainly to the microprocessor core, which is the most complex component on the system, although the methodology could be further generalized to other units, such as memory systems or interconnect. In order to maintain tractability, we divided this problem into two main tasks, as described below.

1.2.1 Performance bug detection

The goal of this task is to develop an automated machine learning-based performance bug detection flow that uses information extracted from legacy designs to tell the designers whether a new design has a performance bug or not.

1.2.2 Performance bug localization

The goal of this task is to expand the methodology implemented for bug detection so that we not only provide the designers with a yes or no answer regarding bug presence, but also provide insight regarding which microarchitectural unit of the processor is the one with the performance bug.

1.3 Thesis statement

This dissertation aims to advance the state-of-the-art automation techniques available for two specific tasks in the performance prediction and diagnosis of VLSI designs area. The first tackles performance at the logic level, in the form of static timing analysis and path slack predictions, the second one, at a higher level, tackles detection and localization of microarchitecture level bugs affecting the total runtime of some applications.

The proposed methodologies for both of these tasks are based on Machine Learning methodologies, which has proven to be an extraordinary method to extract knowledge from data, and handle very complicated non-linear behaviors, making it the best approach to mimic a human manual process among other mathematical or algorithmic options.

1.4 Summary of contributions

Throughout this dissertation, we present several machine-based techniques for performance prediction and diagnosis of VLSI designs, in particular, the main contributions of this work are:

1. The first study on the usage of machine learning for pre-routing timing prediction is conducted [1]. For this study, several machine learning engines are investigated, and a set of input features required for the timing prediction is obtained. The results of this study achieve

pre-routing timing predictions with an accuracy that is comparable to post-routing sign-off timers.

2. The first study on the usage of machine learning for automatic detection of performance bugs in microprocessor cores [2]. For this study, multiple strategies and machine learning techniques are evaluated, and it is found that a two-stage approach provides the best results. This method detects 91.5% the evaluated core performance bugs that lead to a $\geq 1\%$ IPC degradation with 0% false positives.

3. The first study on the usage of machine learning for automatic localization of performance bugs in microprocessor cores. For this study, multiple methodologies are evaluated, and it is found that the best performing methodology identifies the correct bug location as the most likely option in $\sim 75\%$ of the cases and achieves over 98% accuracy when a list of the three most likely options (out of 16 possible) is provided for bugs with an average IPC impact of $>1\%$. Further, for bugs with impact over 0.1% the top-3 accuracy is over 75%. Moreover, our techniques can handle false alarms generated by bug detection.

2. PRE-ROUTING TIMING PREDICTION [1]

2.1 Background

2.1.1 Integrated circuits design flow

As the number of transistors that manufacturers are able to fit into a chip grows, the complexity of the flow required to design an integrated circuit increases. Due to this complexity, the IC industry heavily relies on the usage of design automation software.

Figure 2.1 shows a diagram of simplified design flow for digital circuits, keeping only the most important steps. Even though the diagram shows a unidirectional flow, the design process usually requires iterating back over several steps, until a solution meeting all the design constraints and specifications is obtained.

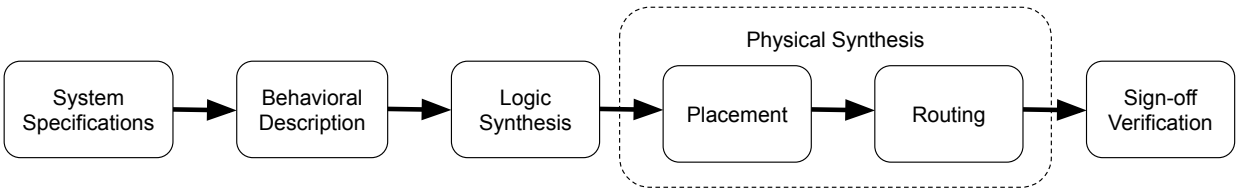


Figure 2.1: Simplified diagram of the design flow for integrated circuits.

Each of the steps in Figure 2.1 corresponds to a complex task, which involves several sub-procedures that are not covered in detail in this dissertation, as they are out of the scope of this work. However, a brief description of these steps is provided below:

1. **System Specifications:** This step defines the design functionality, provides a list of features that must be included in the design, and technical specifications such as clock frequency,

[1] ©2019 IEEE. Reprinted, with permission from E. Carvajal Barboza, N. Shukla, Y. Chen and J. Hu, "Machine Learning-Based Pre-Routing Timing Prediction with Reduced Pessimism", *ACM/IEEE Design Automation Conference (DAC)*, 2019.

power consumption, and several others depending on the purpose of the design and its target market.

2. **Behavioral Description:** Once the major characteristics of the design have been determined, these are translated into behavioral models of the design, usually written in a hardware description language, such as VHDL or Verilog.

3. **Logic Synthesis:** During this step, the behavioral description of the design is transformed into a digital circuit by listing every logic gate, sequential elements, and their corresponding connectivity. This description is usually known as a "netlist". Due to the high complexity of this procedure, automated tools are usually used for this process. Along with the transformation, several timing, power, and area optimizations also take place at this step [9].

4. **Physical Synthesis:** In this step, the netlist generated by the logic synthesis is further transformed into a physical design by deciding where each logic gate will be placed and what trajectory each interconnection will follow. Both placement and routing are very complex tasks, usually requiring several iterations to achieve a satisfactory result. Several specifications, such as performance, power consumption, and heat distribution rely heavily on the quality of the solution obtained at this stage.

5. **Sign-off Verification:** In this step the design is evaluated using more detailed, and therefore slower models. This allows for better accuracy in the results and higher confidence in the timing, power, and reliability metrics. Once all the checks in this step have passed, the design is ready for manufacturing.

2.1.2 Timing models

Briefly speaking, there are two kinds of timing models that are often employed for logic and physical synthesis of integrated circuits:

1. A sign-off model that evaluates if a circuit design satisfies timing specifications. It estimates gate delay using lookup tables [10] or current source model; these models consider slew-

rates, and the wire delay estimation uses high-order models [11]. Additionally, sign-off timing analysis tools consider many details such as rising/falling switching, crosstalk, false paths, and simultaneous switching.

2. A relatively fast model that is often invoked within synthesis and optimizations. In this case, a gate is modeled as an RC switch, wires are modeled as RC trees, and the delay is computed using the Elmore method [12].

To obtain an overall estimation of the timing in the design, the gate and wire delays are collected using addition/subtraction and max/min operations through PERT traversals [13] to obtain signal arrival time, required arrival time, and slack at each node in a circuit.

For timing estimation at placement, and before routing, gate delay can be obtained using sign-off models while wire delay is greatly simplified because the routing information is not available. For example, the source-sink delay of a net can be estimated using an analytical formula based on their distance [14]. Such pre-routing models are not very accurate, as they do not consider potential wire detours due to congestion avoidance or the impact produced by the metal layer assignment. Moreover, without wire parasitic information, the higher-order interconnect model has no ground to carry out accurate delay computation.

2.2 Prior work

There have been a few previous works aiming to apply machine learning methodologies for timing analysis, but they are targeted for applications in different contexts from our work. For example, the work by Kahng *et al.* in 2014 [15] aims to guide the gate sizing and threshold voltage assignment in post-routing timing recovery and power optimization. For this, a set of machine learning models that estimate wire delay and slew are used on iterative static timing analysis to achieve a more accurate estimation of endpoint slack without the need to constantly re-run the sign-off tool timing models due to adjustments on gate sizing or threshold voltage swapping. In this work, the ML is used to learn weights that are later applied to the delay and slew results obtained by multiple analytical models.

Han *et al.* [16] studied several machine learning techniques, including random forest and SVM to overcome the discrepancy between different commercial timing analysis tools. The proposed methodology correlates path slack between different timing tools across multiple technology nodes and designs, reducing the range of divergence by over $5.5\times$. In particular this approach was evaluated by correlating two different signoff tools, as well as a signoff tool with a design implementation tool.

The work by Kahng *et al.* in 2015 [17] makes use of neural network and SVM techniques to estimate the effect of coupling capacitance in timing analysis without the need to run commercial timing analysis tools in signal integrity mode. This work achieved worst-case error reductions of up to 55ps.

In work done by Chan *et al.* in 2016 [18], boosting and SVM are applied to the floorplanning stage to predict timing failures of embedded memories. The high uniformity of embedded memory circuits allows for such early timing failure predictions, unlike the case of other circuits where the structure is not as well-defined.

More recently, work by Lu [19] uses conditional generative adversarial networks (GAN) to optimize the clock tree synthesis (CTS) outcome via reinforcement learning. This methodology extracts features from placement images and uses a GAN to test combinations of input settings to be used during the clock tree synthesis flow and determine whether they will outperform the designs obtained automatically by the commercial tools.

Overall, there is no previous work on machine learning-based pre-routing timing prediction for circuit optimizations at the placement stage, to the best of our knowledge.

2.3 Machine learning for pre-routing timing prediction

2.3.1 Problem formulation

Given an un-routed, but placed circuit database, the goal of this work is to develop an accurate and fast model that is able to predict the post-route slack of every path in the design. This will aid designers by providing results with post-routing sign-off tool quality at early stages of the design,

preventing the need of multiple iterations or over-designing.

The proposed methodology consists of a machine learning model able to predict net delay using information that is readily available at pre-routing stages. The model performs the predictions on a per-net basis. In other words, the delay model will provide a prediction of the delay between the input pin of a driver gate and the sink pin of the receiver gate. Using Net 'D' of Figure 2.2 as an example, our model will predict the delay to pin 'g', incorporating the delay of driver gate 'U2', as well as the delay due to the wire between pins 'w' and 'g'. In the case of multi-fanout drivers, as is the case of Net 'C', our model will predict two separate delays, one to pin 'e' and another to pin 'f', and just as in the previous case, both of these timing predictions will incorporate the delay of the driver gate 'U1' and their delay produced by the wire.

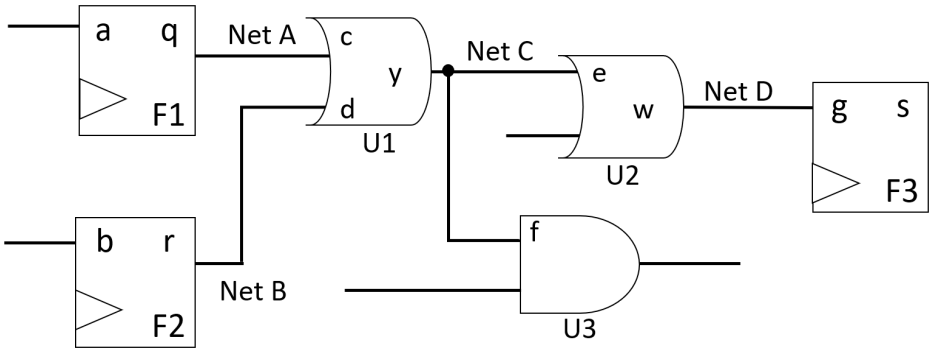


Figure 2.2: An example circuit for timing estimation.

Once the delay of all source-sink pairs has been calculated, PERT traversals [13] are applied on the circuit graph, which is extracted from the design netlist. This is done so that more useful metrics for designers, such as arrival time, required arrival time, and path slack can be obtained.

Our model does not differentiate among the different input pins of a single driver gate (pins 'c' and 'd' of cell 'U1'). The different arrival times of the pins are considered only during the PERT traversal. This strategy allows our model to perform a single prediction for multi-input gates with the expense of minor approximation. For similar reasons, our model does not differentiate between

rising and falling delays, it only predicts the delay for the worst case.

2.3.2 Methodology overview

An overview of the proposed methodology is depicted in Figure 2.3. In the figure, the blue boxes correspond to inputs to our methodology, obtained in the pre-routing stages of the design flow. The red boxes indicate labeled data, which is required only during training phases. The gray box indicates our final output.

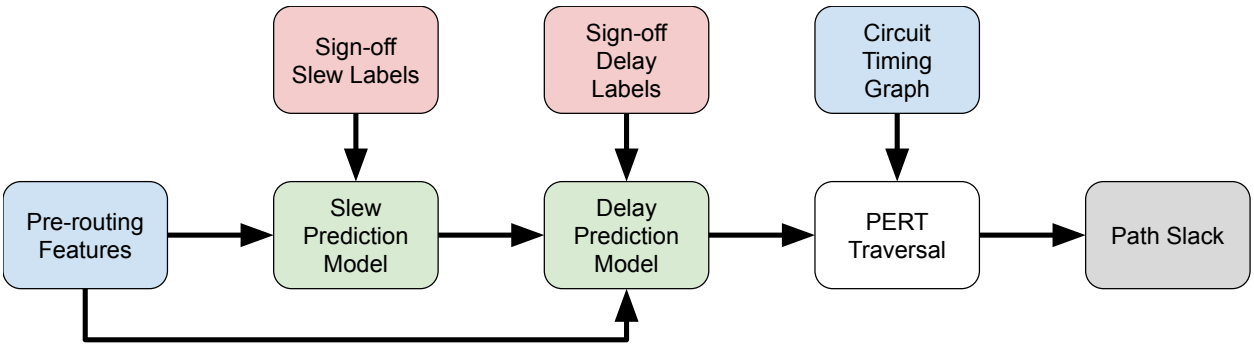


Figure 2.3: Timing prediction methodology structure flow.

The proposed methodology takes as input features a set of data extracted from pre-routing databases, a detailed explanation of the features used as inputs to the ML models implemented for this methodology is given in Section 2.3.3. The features are used in two different machine learning models:

- **Slew prediction model:** We define the slew rate as the signal transmission time, which means that a sharp signal transition will have small slew. Slew information is vital for an accurate timing estimation, but it usually requires a thorough timing analysis in order to produce accurate results. For this reason, we developed a dedicated net slew model, which predicts the slew rate at every sink pin by using the pre-routing features. This model is trained using labeled data obtained with a sign-off timer tool.

- **Delay prediction model:** This model takes the pre-routing features along with the information predicted by the slew model mentioned above. This model is the one in charge of generating the delay predictions for every net on the design. To train it, labeled data obtained with a sign-off timer tool is used.

Once the delay predictions of every net in the design have been performed, a circuit timing graph, extracted from the design netlist, is used in order to perform the PERT traversals [13]. This method will then produce the path slack information, which is ultimately the most important metric for circuit designers.

2.3.3 Model features

The pre-routing features correspond to the data that will be used as inputs to the machine learning model so that a prediction can be performed. A good set of input features is critical for the accuracy of the model. The features used for the delay model are elaborated as follows:

- **Driver capacitance:** The driver output capacitance is proportional to its driving strength. Larger values indicate faster drivers and therefore smaller delays. This value can be extracted from the standard cell library being used for the design.

- **Sink capacitance:** The sink input pin capacitance is proportional to the load seen by the driver. Larger values indicate more load and therefore larger delays. The proposed model uses two different features to represent this information:

    1. The **target sink capacitance**, which corresponds only to the capacitance of the input pin to which the delay is being predicted.

    2. The **total sink capacitance**, which corresponds to the sum of the input capacitance of every sink pin connected to the net.

    Going back to the example of Figure 2.2, when predicting the delay to pin ‘e’ in Net ‘C’, the target sink capacitance will correspond only to the capacitance provided by pin ‘e’, while the

total sink capacitance will also include the capacitance added to Net 'C' due to pin 'f'. This value can also be extracted from the standard cell library that is used to generate the design

- **Distance between driver and target sink:** The horizontal and vertical distances between the driver gate and the target sink are proportional to the wire delay, especially when buffers are inserted in long nets [14].

- **Context sink locations:** In multi-fanout nets, when the proposed model is applied to determine the delay between the source gate and the target sink, every other sink connected to the same net is called a "context sink". As an example, in Net 'C' of Figure 2.2, when the delay to sink pin 'e' is being predicted, sink pin 'f' serves as a context sink. These context sinks not only contribute to the total load capacitance, but their locations can also affect routing, buffering, and therefore delay to the target sink. Since the number of context sinks varies among nets, but the evaluated machine learning models require a fixed size input, the context sink locations are approximated by providing the model with statistical signatures. One is the median position of all context sink, which provides the model with information of roughly, how far away these are from the driver. Another one is the standard deviation of the locations in both dimensions, this indicates to the model how spread they are, which correlates with the corresponding interconnect tree size.

- **Driver input slew:** The slew information is obtained via a dedicated machine learning model, as explained in Section 2.3.2. As the number of input pins varies from gate to gate, and the evaluated machine learning models require a fixed input size, only the maximum slew across every input pin of the driver gate is considered. For example, when the slew model is applied to Net 'C' at Figure 2.2, it will provide estimations for sink pins 'e' and 'f', these will later be used to estimate the delays of Net 'D' and the net driven by gate 'U3'.

2.3.4 Data extraction

In order to extract the data required for the training and testing of this methodology the flow described in Figure 2.4 is followed.

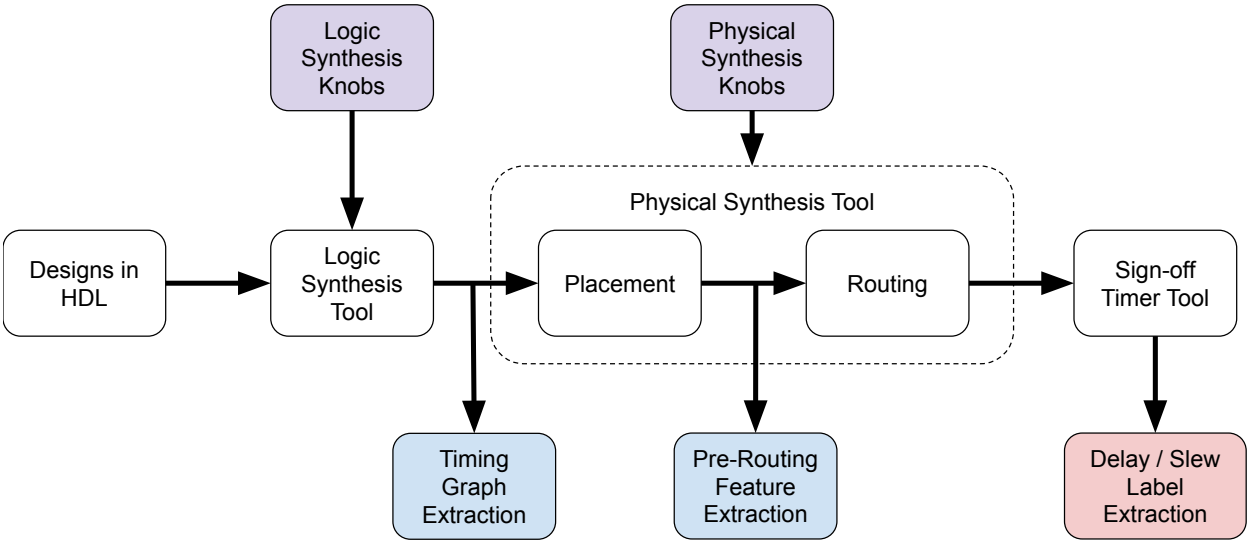


Figure 2.4: Timing prediction data extraction flow.

In detail, the steps are described below:

1. Designs are described in a behavioral manner in a Hardware Description Language, such as Verilog or VHDL.

2. Using a commercial logic synthesis tool, the behavioral description is transformed into a netlist, which enumerates the logic gates, sequential elements, and their connections. To augment the dataset, the logic synthesis tool is executed multiple times with different values on some of the tool knobs such as mapping effort, dynamic power optimization, static power optimization, clock gating, maximum fanout, maximum load capacitance, maximum transition time, among others. From the resulting netlists, the circuit timing graph that will be used for the PERT traversals is extracted, this is required in order to obtain each path slack.

3. Each of the generated netlists is then processed by a commercial physical synthesis tool, which transforms the netlist into a physical layout design. As in the previous step, the dataset is augmented by running this step multiple times changing some of the tool knobs, such as maximum utilization, congestion effort, timing optimization, signal integrity optimization, among others. Physical synthesis is usually done in two separate steps: placement and

routing. After the first stage is done, the data that will be used as features by the proposed model is extracted. For the intended application usage, once here, and before running the routing, the machine learning model is used to obtain timing estimations, and fixes can be applied by the designers to address the issues with the current implementation. However, to gather training data for the model, the rest of the flow is executed, so that the labels can be extracted.

4. Finally, a sign-off timing analysis tool is executed, and the slew and delay data is extracted, so that they can be used as labels to train the machine learning models.

2.3.5 Customized loss function neural network

When performing timing estimations designers prefer to estimate delays higher than what they actually are, so that they err on the cautious side. For that reason, in this section, a custom loss function in the neural network model is proposed, so that the models learn to follow the same cautious behavior. Commonly, Neural Networks use Mean Squared Error (MSE), described by Equation 2.1, as its loss function.

$$\text{MSE} = \frac{1}{N}\sum_{i=0}^{N}(\hat{y_i} - y_i)^2 \tag{2.1}$$

Here y_i represents the actual value, $\hat{y_i}$ represents the value estimated by the model, while N represents the total number of samples. This metric is symmetric, and the same loss weight will be applied to both positive and negative errors. However, since designers would prefer to have estimations that are higher than the actual value (positive error values), an asymmetric MSE is implemented as described by Equation 2.2.

$$\text{Asymmetric MSE} = \text{MSE} + \frac{1}{2n}\sum_{i=0}^{n}(|\hat{y_i} - y_i| - \hat{y_i} + y_i)^2 \tag{2.2}$$

Note that, for positive errors ($\hat{y_i} \geq y_i$) the loss function is equal to the regular MSE, while for negative errors ($\hat{y_i} < y_i$) the asymmetric MSE is equal to two times the MSE. Thus, this loss

function forces the model to prevent predictions that under-estimate the delay. A comparison on the accuracy of this method is shown in Section 2.5.2

2.4 Experimental setup

The experimentation was performed on benchmark circuits from ITC'99 [20], in particular the selected designs are b11, b12, b13, b14, b17, b20, b21, and b22. The ITC'99 benchmarks are written in VHDL language. These designs go through the logic synthesis procedures using Synopsys Design Compiler, and the generated netlist is then feed to Cadence Encounter to perform the physical synthesis. The physical synthesis includes steps such as cell placement, buffer insertion, and routing. For labeled data generation, a subset of the routing solutions are fed to Synopsys PrimeTime, which is considered a sign-off analysis tool.

By modifying some knobs on the logic and physical synthesis tools, as described in Section 2.3.4, 40 different implementations of each of the b11, b12, b13, b14, b20, and b21 circuits are obtained (240 different circuit implementations in total). From these, 20 of each circuit are used as part of the training set for the ML models (120 different circuit implementations in total). The other half of these, along with 20 different implementations of b17 and b22 are used to test the models (160 implementations in total). This allows to evaluate the performance of the models in designs that the models have seen before, as well as designs that are completely new for the models.

The circuits are implemented using the NanGate 45nm standard cell library [21]. Although the proposed methodology is able to work on designs that have not been seen by the models before, the models learn the behavior of the specific standard cell library on which they are trained, therefore, if a different library is used, the models will need to be re-trained.

The machine learning models were implemented, trained, and tested using Python [22]. The Lasso Linear Regression [23] and Random Forest [24] models were implemented using the Scikit-Learn library [25], while the Neural Networks were implemented using Keras [26]. Besides these machine learning models, a commercial tool on pre-routing timing prediction is used to compare the obtained results.

2.5 Results

This section presents the results of multiple evaluations performed to the ML-based timing estimation methodology proposed here.

2.5.1 Net slew and delay estimation quality

In this section, the quality of the machine learning models on individual net delay and slew prediction is evaluated. Please note that testing data includes 20 different implementations of designs b11, b12, b13, b14, b20, and b21; although the model was trained using these designs, every implementation used for testing is slightly different to the ones used for training. All 20 implementations of designs b17 and b22 are completely new to the model since these designs were not included for the training.

The evaluated machine learning engines are Lasso Linear Regression [23], Random Forest [24] and Artificial Neural Networks [27]. In the case of the latter, two different variants were tested, the first one uses Mean Squared Error (MSE) as its loss function, while the second variation uses the custom loss function explained in Section 2.3.5. The results obtained on the testing designs for different machine learning engines are presented on Table 2.1.

Table 2.1: Net slew and delay prediction results for multiple machine learning engines.

	Slew prediction		**Delay prediction**	
Method	**Correlation**	**MSE**	**Correlation**	**MSE**
Lasso Regression	0.739	1.4×10^{-3}	0.710	1.9×10^{-3}
Neural Network (MSE)	0.861	7.6×10^{-4}	0.891	5.7×10^{-4}
Neural Network (Custom)	0.856	8.1×10^{-4}	0.882	6.1×10^{-4}
Random Forest	0.974	1.0×10^{-4}	0.971	1.8×10^{-4}
Commercial tool	0.909	3.8×10^{-4}	0.947	3.7×10^{-3}

On Table 2.1, both Pearson's correlation factor and MSE are calculated with respect to the post-routing analysis obtained using Synopsys PrimeTime, as that metric is considered the "Ground Truth". These results are averaged across all testing circuits and weighted by the number of end-

points, *i.e* the number of flip-flops and primary outputs. Therefore, large circuits carry relatively large weights.

The results show that the proposed models for slew prediction have an accuracy on-par with the estimations provided by the pre-routing commercial tool. However, the pre-routing commercial tool has a much larger MSE than the obtained by the proposed machine learning models in terms of delay prediction. In particular, it can be seen that the proposed method produces the least MSE and highest correlation when Random Forest is used. Not only is Random Forest far more accurate than the other machine learning engines evaluated, but the achieved error is an order of magnitude lower than the error from the pre-routing commercial tool. A detailed circuit by circuit comparison between the Random Forest model and the commercial tool is provided in Table 2.2.

Table 2.2: Comparison of net skew and delay prediction results between random forest and a commercial tool.

			Slew prediction				**Delay prediction**			
			Random forest		**Commercial tool**		**Random forest**		**Commercial tool**	
Circuit	**# nets**	**# FFs**	**MSE**	**Correlation**	**MSE**	**Correlation**	**MSE**	**Correlation**	**MSE**	**Correlation**
b11	926	30	1.69×10^{-6}	0.998	2.29×10^{-5}	0.976	6.18×10^{-5}	0.986	1.71×10^{-3}	0.966
b12	2409	121	1.69×10^{-6}	0.999	1.67×10^{-4}	0.948	9.71×10^{-5}	0.968	1.84×10^{-3}	0.992
b13	586	53	1.42×10^{-6}	0.999	3.69×10^{-5}	0.979	9.62×10^{-5}	0.954	1.54×10^{-3}	0.987
b14	28828	247	5.25×10^{-6}	0.999	6.49×10^{-4}	0.854	7.57×10^{-5}	0.989	4.02×10^{-3}	0.967
b17	31084	1407	1.11×10^{-4}	0.962	1.79×10^{-4}	0.919	3.39×10^{-4}	0.924	2.91×10^{-3}	0.980
b20	18907	494	9.62×10^{-5}	0.978	4.48×10^{-4}	0.912	1.41×10^{-4}	0.980	3.44×10^{-3}	0.937
b21	25034	494	1.16×10^{-4}	0.967	3.83×10^{-4}	0.937	2.09×10^{-4}	0.972	3.41×10^{-3}	0.967
b22	45354	709	1.15×10^{-4}	0.974	3.72×10^{-4}	0.910	1.72×10^{-4}	0.976	4.72×10^{-3}	0.922

Results on Table 2.2 show that the slew predictions of the random forest model are slightly better than the commercial tool estimations in every circuit, both for correlation, as well as MSE. In terms of net delay prediction, the random forest model achieves MSE that are at least one order of magnitude smaller than the commercial tool, even on designs b17 and b22, which were not used for training.

A histogram depicting the distribution of the error on net delay predictions is shown in Figure 2.5. The figure clearly shows the advantage of the proposed random forest model over the commercial tool. The random forest predictions have not only smaller errors but also a smaller

variation. By contrast, the estimations by the commercial tool are very pessimistic and the errors are largely spread.

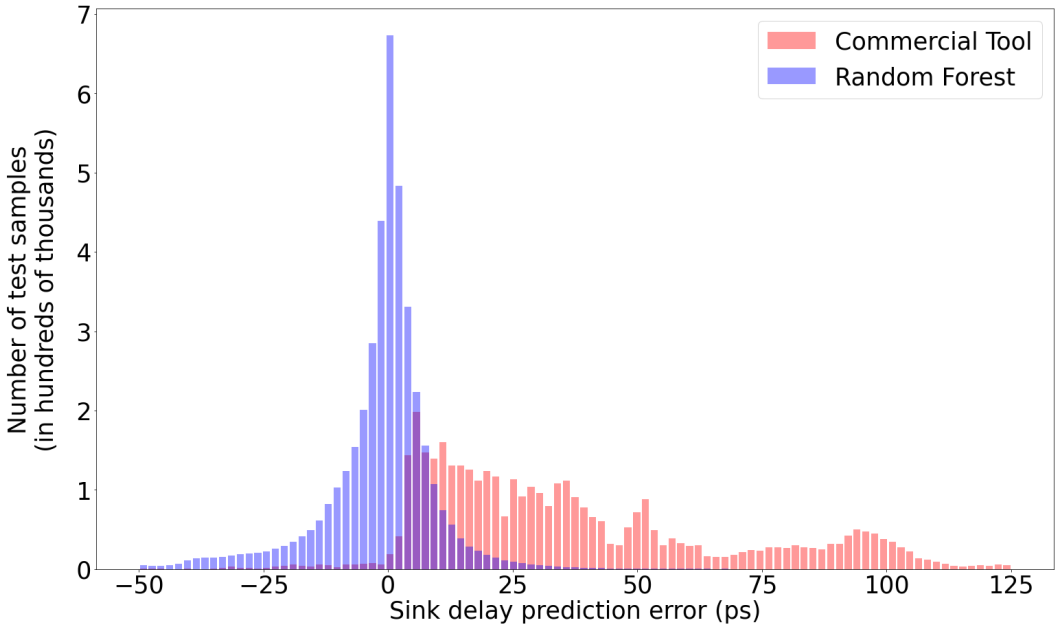
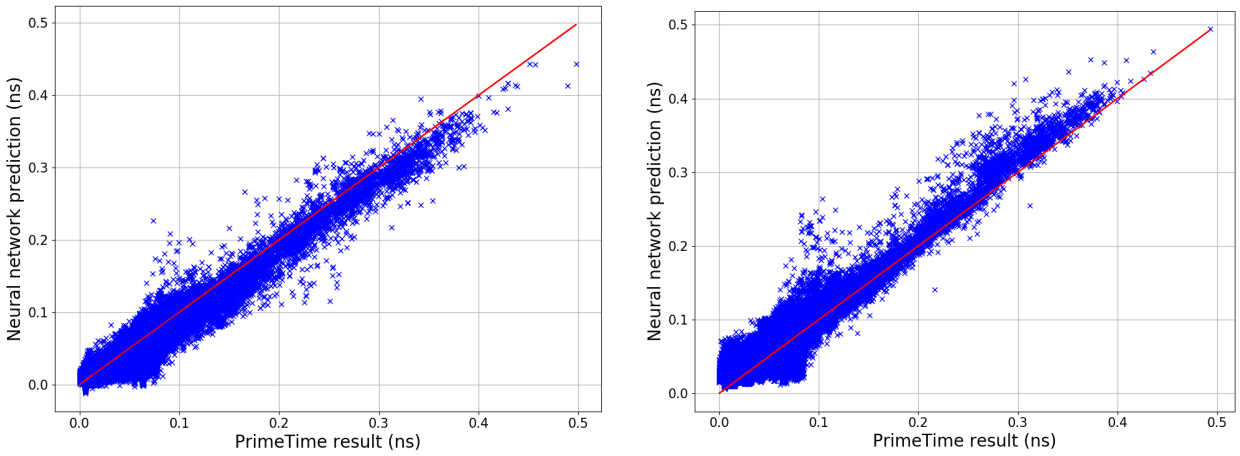


Figure 2.5: Error distribution of sink delay predictions.

2.5.2 Impact of the customized loss function on the neural network prediction accuracy

To evaluate the impact of using the asymmetric MSE as loss function, both variations of Neural Networks (regular MSE and custom MSE) were trained and tested using identical network architectures and well as identical train and test sets. The only difference between them is the loss function. The results of this experiment are shown in Figure 2.6. In the figure, the x-axis shows the net delay as calculated by PrimeTime, a post-routing sign-off timer tool, the y-axis shows the delay estimated using the corresponding NN. Each dot in the figure represents a different source-sink pair. Therefore, the red line indicates a perfect match between the ML prediction and the post-routing sign-off analysis.

It can be seen that the custom loss function forces the predictions to shift upwards, and therefore reduces the number of under-estimated delays. From the 90,010 source-sink pairs on the test set,



(a) MSE as loss function

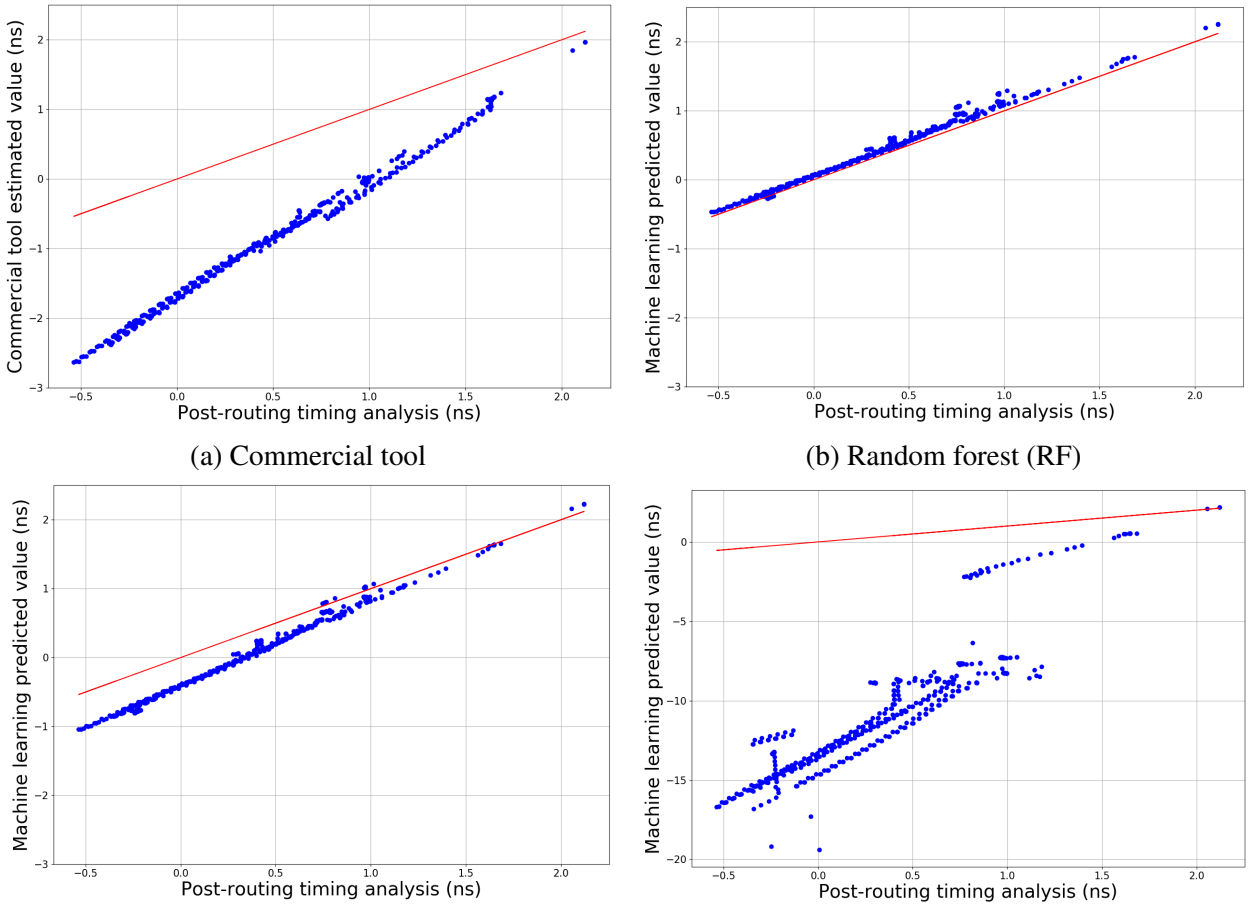
(b) Asymmetric MSE as loss function

Figure 2.6: Neural network prediction versus PrimeTime results on net sink delay.

only a 22.06% is underestimated when the asymmetric loss function is used, while a 75.26% is underestimated when the regular MSE is used as loss function. Despite the improvement from the custom loss function, the overall accuracy achieved by the neural network model is not as good as that of random forest-based model. This method was evaluated exclusively on Neural Networks because the libraries implementing the other ML engines do not allow for custom loss functions.

2.5.3 Path slack prediction accuracy

Although having accurate predictions of the delay of every net in the design is important, even more significant is to have accurate predictions of the slack available at each endpoint in a design. This is because, ultimately this will be the determining factor to identify timing violations. To illustrate this, endpoint (either flip-flops or primary outputs) slack predictions from different methods are compared for one implementation of circuit b22 and the results are plotted in Figure 2.7. In the figure, the x-axis shows the path slack as calculated by PrimeTime, which is considered the "ground-truth", since it is a post-routing sign-off timer tool, the y-axis shows the slack predicted by using the corresponding ML engine. Each dot in the figure represents a different endpoint. Therefore, the red line indicates a perfect match between the ML prediction and the post-routing sign-off analysis estimation.



(a) Commercial tool

(b) Random forest (RF)

(c) RF with correction

(d) NN custom loss

Figure 2.7: Path slack predictions of circuit "b22" for different methods compared to sign-off timer (PrimeTime) results.

Note that circuit "b22" is not included in the training data, and therefore none of the nets of this design were used to train the model. The goal of this plot is to illustrate slack predictions on different paths of a design, and how accurate the ML engines can be. Although here we show only results for one implementation of "b22", the results on other circuits and implementations are similar to these.

Figure 2.7a shows that the pre-routing commercial tool estimations are quite pessimistic for slack prediction, its MSE across this design is 2.023. Figure 2.7b shows that the random forest-based prediction is much closer to the perfect match with a MSE of 0.012, which is more than $150\times$ smaller than the commercial tool.

However, one disadvantage of the random forest model is that, as it can be seen in Figure 2.7b, there are multiple paths for which the predicted slack is greater than what it is in reality, which could lead to functional issues when the circuit runs at the specified speed. To avoid this, a scaling-based correction was implemented. What this methodology does is simply to multiply the obtained maximum arrival time of each endpoint by a factor $\alpha > 1$, which will make the predictions larger, and therefore the slack predictions become more pessimistic, pushing down the slack curve as shown in Figure 2.7c for $\alpha = 1.15$. Even after this correction, the MSE is still at 0.114, which is still $> 15\times$ lower than that from the commercial tool. The result in Figure 2.7d confirms that neural network is quite inferior in slack prediction.

The overall slack predictions results are summarized in Table 2.3 for PrimeTime (which constitutes the "Ground Truth"), the Random Forest models (both with and without scaling factor), as well as the pre-route commercial tool. The quality of the results is measured by comparing the Total Negative Slack (TNS). It can be seen that the result from random forest prediction is much closer to PrimeTime than the commercial tool prediction. The error of TNS prediction is consistently one order of magnitude lower than the pre-route commercial tool on every circuit. The results normalized against the PrimeTime TNS are also shown in this table.

Table 2.3: Results on total negative slack separated by circuit design on the benchmark.

	PrimeTime	**Random forest**		**Random forest w/ correction**		**Commercial tool**	
Circuit	**TNS (ns)**	**TNS (ns)**	**Normalized TNS MSE**	**TNS (ns)**	**Normalized TNS MSE**	**TNS (ns)**	**Normalized TNS MSE**
b11	-0.14	-0.49	3.50	-1.37	9.79	-5.33	38.07
b12	-0.90	-2.94	3.27	-5.51	6.12	-22.42	24.91
b13	-0.04	-0.35	8.75	-1.20	30.00	-5.78	144.50
b14	-9.05	-12.01	1.33	-40.37	4.46	-143.86	15.90
b17	-7.73	-0.56	0.07	-10.14	1.31	-367.79	47.58
b20	-48.46	-69.38	1.43	-151.63	3.13	-412.09	8.50
b21	-17.48	-27.82	1.59	-85.71	4.90	-299.29	17.12
b22	-22.23	-18.08	0.81	-68.21	3.07	-383.27	17.24
Average	**-13.25**	**-16.45**	**1.24**	**-45.52**	**3.44**	**-204.98**	**15.47**

In terms of runtime, on average, across the 160 design implementations (20 implementations per circuit $\times$ 8 circuits) for testing, the random forest-based timing prediction is $30\times$ faster than PrimeTime and only $3\times$ slower than the commercial pre-routing timing estimation.

2.5.4 Critical path classification accuracy

The most important application of timing analysis or slack prediction is to identify the timing critical paths where additional optimization effort is needed. In practice, designers often apply additional guard-band to avoid missing any potentially critical paths. For example, a path is considered critical when its slack is smaller than a threshold $\theta > 0$. In this experiment, a $\theta = 15ps$ is used and then machine learning prediction is applied in order to classify if the path with minimum slack at each endpoint is critical or not. The following are a couple of commonly used metrics for evaluating classification accuracy.

- TPR (True Positive Rate): the ratio of the number of truly critical paths correctly identified by a classifier versus total number of truly critical paths. This captures the percentage of actual critical paths the method is able to detect. A perfect TPR is 1.
- TNR (True Negative Rate): the ratio of the number of truly non-critical paths correctly identified by a classifier versus total number of truly non-critical paths. A perfect TNR is 1.
- FPR (False Positive Rate): 1 - TNR.
- FNR (False Negative Rate): 1 - TPR.

The TPR and TNR results from random forest-based classification (with and without correction) and the pre-route estimations provided by the commercial tool are shown in Table 2.4. It can be seen that random forest reaches TPR of 1 for all circuits except b17. There could be two reasons for this: (1) b17 is not part of the training data, therefore results with slightly less accuracy are expected or (2) b17 is an intrinsically difficult case, the large errors obtained by the commercial tool hint that the second reason might be true.

Table 2.4: Results on circuit slack and critical path classification divided by circuit design on the benchmark.

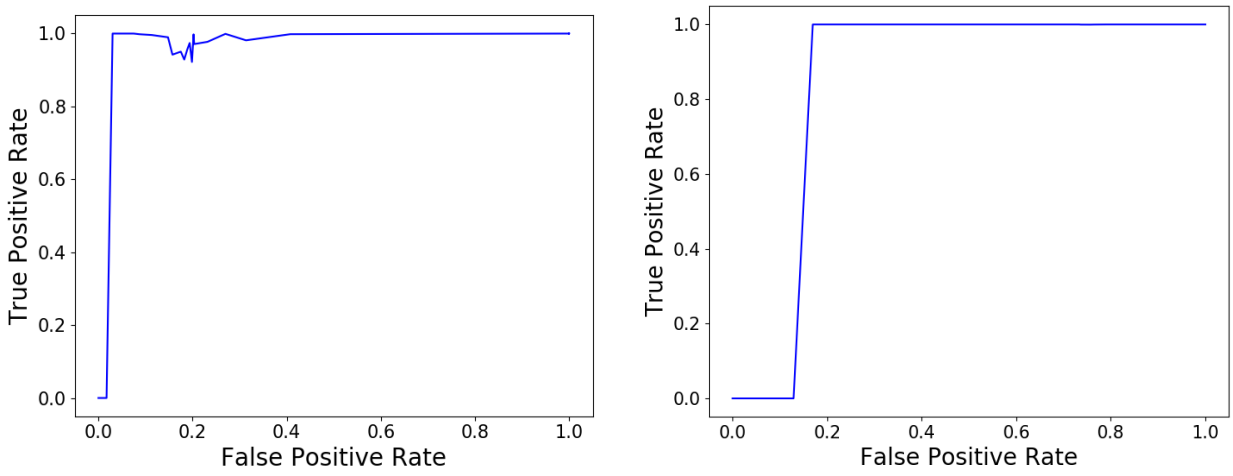
	PrimeTime	Random forest				Random forest w/ correction				Commercial tool			
Circuit	**# critical paths**	**# critical paths**	**TPR**	**TNR**	**ROC area**	**# critical paths**	**TPR**	**TNR**	**ROC area**	**# critical paths**	**TPR**	**TNR**	**ROC area**
b11	4	8	1.00	0.86	0.94	15	1.00	0.55	0.74	28	1.00	0.02	0.29
b12	24	34	1.00	0.90	0.98	41	1.00	0.81	0.94	114	1.00	0.96	0.72
b13	4	10	0.97	0.84	0.93	26	1.00	0.43	0.75	42	1.00	0.03	0.37
b14	39	45	1.00	0.97	0.99	78	1.00	0.77	0.90	150	1.00	0.36	0.71
b17	49	10	0.19	1.0	0.50	57	0.92	0.99	0.89	759	1.00	0.44	0.80
b20	116	166	1.00	0.86	0.99	166	1.00	0.61	0.90	330	1.00	0.31	0.68
b21	53	93	1.00	0.90	0.99	184	1.00	0.66	0.94	343	1.00	0.33	0.69
b22	73	74	0.89	0.99	0.94	148	1.00	0.86	0.96	459	1.00	0.32	0.63
Average	**45**	**55**	**0.88**	**0.91**	**0.91**	**99**	**0.99**	**0.71**	**0.88**	**278**	**1.00**	**0.35**	**0.61**
Weighted Average	**-**	**-**	**0.86**	**0.96**	**0.98**	-	**0.99**	**0.85**	**0.97**	**-**	**1.00**	**0.38**	**0.85**

It can also be observed that the commercial tool has much lower TNR than random forest. Its TNR results indicate that the number of false alarms is usually over 60%, which lead to considerable over-design and wastes precious engineer time. The random forest-based classification achieves satisfactory results, even on design b22, which is not included in the training data. The results in Table 2.4 also confirm the importance of the correction for random forest model. Without the correction, the TPR of random forest b17 is very low. The average results in the last row of Table 2.4 are weighted according to the number of endpoints of each circuit. As such, large circuits carry more weight in the average results.

Evidently, TPR and TNR are affected by the value of threshold θ. When θ becomes large, TPR increases and TNR decreases. The Receiver Operating Characteristic (ROC) curve is a way to evaluate classification accuracy for different threshold values. In ROC curves, the x-axis shows the FPR, while the y-axis indicates the TPR. A perfect classifier would reside on the point $(0, 1)$. Figure 2.8 shows the ROC curves for the classifications obtained using random forest with correction (Figure 2.8a) as well as the pre-route commercial tool (Figure 2.8b). The oscillations in Figure 2.8a are due to the fact that the slack estimate of the machine learning model is occasionally optimistic.

An overall numeric metric for an ROC curve is the area under the ROC curve. A random guess would obtain an ROC area of 0.5, while a perfect classifier can achieve ROC area 1. The ROC area

results for all test cases are also listed in Table 2.4. The ROC areas from random forest is 0.97 in the weighted average, which is remarkably higher than the 0.85 from the commercial tool.



(a) Random forest with correction

(b) The commercial tool

Figure 2.8: ROC curves for critical path classification.

Another purpose of the timing prediction is to identify non-critical paths, which are those from which timing can be sacrificed in order to reduce power consumption. Experiments were conducted to identify those paths with a slack $> 100ps$. In this classification, the random forest model with correction can correctly identify 79% of the truly non-critical paths while the pre-route commercial tool can identify only 34% of them. Finding more non-critical paths will help the designer with considerable power savings.

2.5.5 Implementation quality ranking

Another application of the timing prediction is to compare design implementations of the same circuit and determine which implementation is superior. In this experiment, the TNS is used to rank the 20 implementations of each circuit. Table 2.5 shows the ranking of the highest ranked implementation (with respect to the actual TNS obtained using PrimeTime) that can be found on the top-3 ranked of either random forest or the pre-route commercial tool. The random forest-

based ranking can often pick the actual best implementation among its top-3 choices, including b17, which is not used in training data. On average, the ranking results from the random forest model is better than that from the commercial tool.

Table 2.5: Best rank circuit implementation found in estimated top-3 using pre-routing ML estimation method.

Circuit	**Commercial tool**	**Random forest with correction**
b11	1st	1st
b12	1st	1st
b13	4th	1st
b14	7th	4th
b17	2nd	1st
b20	3rd	1st
b21	10th	11th
b22	2nd	4th
Average	4th	3rd

2.6 Conclusion

In this chapter, the first systematic study, to the best of our knowledge, on a machine learning approach for pre-routing timing prediction, was presented and evaluated. Model feature selection and several different machine learning engines were investigated. Extensive experiments on benchmark circuits show that the proposed machine learning model can largely reduce the pessimism and improve the accuracy of pre-routing prediction compared to a commercial tool.

3. PERFORMANCE DEBUGGING OF MICROPROCESSOR CORES [1]

3.1 Background

Verification and validation are typically the largest components of the design effort of a new microprocessor core. The effort can be broadly divided into two distinct disciplines, namely, functional and performance verification. The former is concerned with design correctness and has rightfully received significant attention in the research community. Even though challenging, it benefits from the availability of known correct output to compare against.

Alternately, performance verification, which is typically concerned with generation-over-generation workload performance improvement, suffers from the lack of a known correct output to check against. Given the complexity of computer systems, it is often extremely difficult to accurately predict the expected performance of a given design on a given workload. Performance bugs, nevertheless, are a critical concern as the cadence of process technology scaling slows, and they rob processor designs of the primary performance and efficiency gains to be had through an improved microarchitecture.

The main challenge in performance verification is that, unlike functional verification, there are no exact golden references to compare against, and even when available, their performance estimates will typically vary greatly from real system performance [5, 6, 7, 28]. This is because it remains very difficult to model all the interactions between different microarchitectural units in the highly complex microprocessor designs and therefore, it is hard to accurately represent how they affect the overall system performance.

In addition to that, performance debugging suffers from limited visibility on intermediate points, exposed mostly by performance counters, and the lack of a good debugging infrastructure. Currently, performance verification is conducted mostly through manual techniques that rely on rough estimates of performance gain expected by microarchitectural changes [4]. Such manual

[1] ©2021 IEEE. Reprinted, with permission from E. Carvajal Barboza, S. Jacob, M. Ketkar, M. Kishinevsky, P. Gratz and J. Hu, "Automatic Microprocessor Performance Bug Detection", *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2021

processes are not only very lengthy but also error-prone.

A well-documented example of a performance bug is the case of the Xeon Platinum 8160 processor snoop filter eviction bug [8]. Here the performance regression was greater than 10% on several benchmarks. The bug took several months of debugging effort before being fully characterized. In a competitive environment where time to market and performance are essential, there is a critical need for new, more automated mechanisms for performance debugging. Especially with recent designs from Intel [29], AMD [30], ARM [31], and others that have placed a greater emphasis on core performance, scaling dramatically core complexity, and therefore scaling the difficulty in all forms of verification.

The process of performance verification and debugging can be broadly divided into three steps:

1. **Bug detection**: In this step, it is determined whether the design achieves the expected performance or not. If not, then a performance bug is likely to be present on the design.

2. **Bug localization**: In this step, the microarchitectural units causing the performance issues are identified, this will guide designers on where should they focus to fix the bug.

3. **Root-cause analysis**: In this step, the mechanisms producing the bug are fully characterized and a possible fix is isolated.

Performance bugs, while potentially significant versus a bug-free version of the given microarchitecture, might be smaller than the difference between microarchitectures in some cases. Consider the data shown in Figure 3.1. The figure shows the performance speedup on several SPEC CPU2006 benchmarks [32] on gem5 [33], configure to emulate the Intel Core microarchitectures Skylake (6^{th} gen [34]) normalized against Ivy Bridge (3^{rd} gen Intel "Core" [35])

The figure shows that a bug-free Skylake microarchitecture provides nearly $1.7\times$ the performance of the older Ivy Bridge, as expected given the greater core execution resources provided.

In addition to the bug-free case, we collected performance data for two arbitrarily selected bug-cases in the Skylake microarchitecture simulation. *Bug 1* is an instruction scheduling bug wherein *xor* instructions are the only instructions selected to be issued when they are the oldest instruction

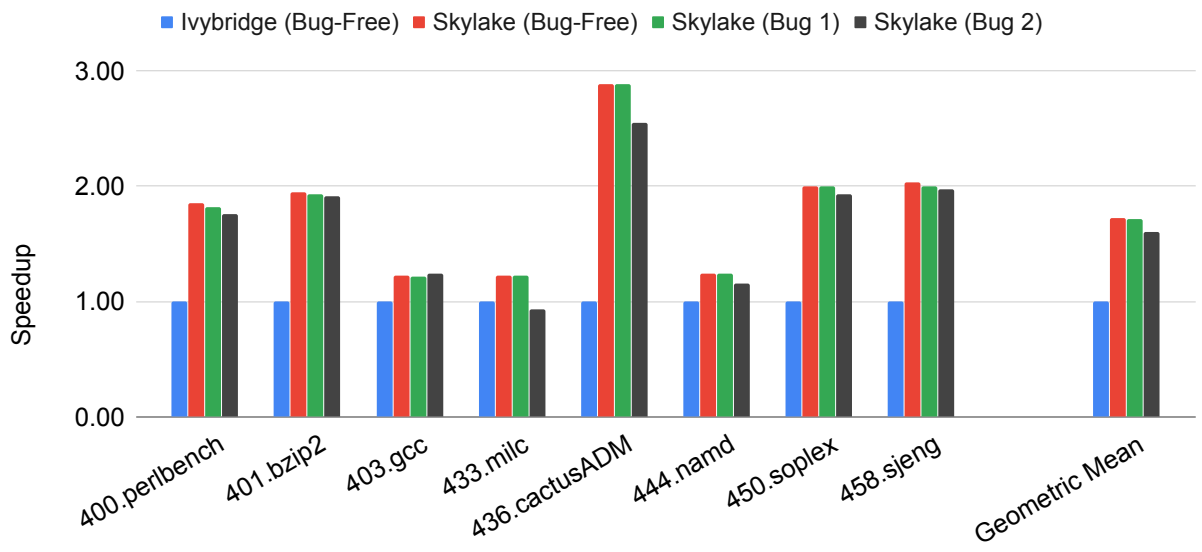


Figure 3.1: Speedup of Skylake simulation with and without performance bugs, normalized against Ivy Bridge simulation.

in the instruction queue, even though other instructions might also be ready to be issued and there are enough resources in the processor to allow it. As the figure shows, the overall impact of this bug is very low (less than 1% on average across the given workloads). *Bug 2* is another instruction scheduling bug which causes *sub* instructions to be incorrectly marked as synchronizing, thus all younger instructions must wait to be issued after any *sub* instruction has been issued and all older instructions must issue before any *sub* instruction, regardless of dependencies and hardware resources. The impact of this bug is larger, at $\sim 7\%$ across the evaluated workloads.

Nevertheless, for *both* bugs, the performance impact is less than the difference between Skylake and Ivy Bridge. Thus, from the designers perspective, working on the newer Skylake design with Ivy Bridge in hand, if early performance results showed the results shown for *Bug 2* in the figure (or even more so *Bug 1*) the designers might incorrectly assume that the buggy Skylake design did not have performance bugs since it was much better than previous generations. Even as the performance difference between microarchitecture generations decreases, differentiating between performance bugs and expected behavior will remain a great challenge given the variability between simulated performance and real systems.

Examining the figure, in can be observed that each bug's impact varies from workload to workload. That said, in the case of *Bug 1*, no single benchmark shows a performance loss of more than 1%. This makes sense since the bug only impacts the performance of applications that may many of the otherwise relatively rare, *xor* instructions. From this, it might be concluded that *Bug 1* is trivial and perhaps needs not to be fixed at all. This assumption, however, would be wrong, as some workloads do rely heavily upon the *xor* instruction, enough so that a moderate impact on them translates into significant performance impact overall.

As an early work in this space, we limit our scope somewhat to maintain tractability. This work is focused mainly on the processor core (the most complex component in the system), however, the proposed methodologies can be generalized to other units such as memory systems. Further, here the focus is on performance bugs that affect cycle count. Circuit-level timing issues in the form of clock period impact are considered to be out of scope. In this dissertation, the proposed methodologies are evaluated exclusively in a pre-silicon setup, however, these techniques could be easily adapted for its usage in post-silicon debug.

Finally, although there may be challenges in validating new microarchitectural techniques, major generalized microarchitectural changes are increasingly rare (*e.g.* the transition from P4 to "core" occurred $\sim$15 years ago). Since then, the generational change in microarchitectures has been largely evolutionary, thus the methodologies proposed here could provide value.

3.2 Prior work

In this section, we discuss the relevant prior work for the performance debug task that we are tackling. Section 3.2.1 discusses some works on performance modeling via analytical and ML-based methodologies. Although relevant for our work, most of these strategies are not applicable for performance debug due to their complexity or due to being unfit for the task. Section 3.2.2 touches on a few works that have used performance counters as inputs to models for power consumption. As our work relies heavily on the usage of performance counters for performance debug, we discuss some of these works here. Sections 3.2.3 and 3.2.4 discuss the few works on performance bug detection and localization, respectively. Finally, Section 3.2.5 discusses other areas

where performance debug is applied, besides microprocessor cores.

3.2.1 Performance modelling

A multitude of works have developed analytical performance models for superscalar out-of-order processors [36, 37, 38, 39, 40] that could be used for performance debugging. However, in order to achieve high accuracy, a large amount of parameters and combination of events that could affect the performance need to be accommodated, which makes analytical models very complex, hence, although useful, these techniques are not widely used for performance verification. As an attempt to automate the performance model generation Ould *et al.* [41] developed a tree-based strategy that uses performance counters in order to estimate the design performance. However, the construction of the model is performed by executing and monitoring the results of multiple traces on the design for which the model is being built, which makes this strategy unfit for performance debugging, as the models will reflect whatever bug present in the design.

3.2.2 Performance counters for power prediction

Prior work has used performance counters to predict power consumption [42, 43, 44, 45]. Joseph *et al.* [42] aim to predict average power consumption on complete workloads. Counters were selected based on heuristics, without automation. Contreras *et al.* [43] further improve this work and present an automated performance counter selection technique that is also able to do time-series prediction of the power. However, they evaluated this technique on an Intel PXA255 processor, which is a single-issue machine, making the problem much simpler than aiming at super-scalar machines. Bircher *et al.* [44] further improves this line of work by creating models not only for the core but also memory, disk and interconnect. The main drawback of this work is that it requires a thorough study of the design characteristics to create the performance counter list to be used. More recently, work by Yang *et al.* [45] further expanded Bircher's work by aiming to develop a full system power model, prior work had only focused on component-based modeling.

3.2.3 Microprocessor performance bug detection

The importance as well as the challenge of processor performance debug was recognized over 25 years ago [46, 47]. However, the follow-up study has been scarce, perhaps due to the difficulty of this problem.

The few known works on bug detection [4, 46, 47, 41] generally follow the same strategy although applying a different emphasis. That is, an architecture model or an architecture performance model is constructed and is then compared with the new design by executing a set of applications. With that, performance bugs can be detected if a performance discrepancy is observed in the comparison. In prior work by Bose [46], functional unit and instruction-level models are developed as golden references. However, a performance bug often manifests in the interactions among different components or different instructions, which this approach is unable to capture. Surya *et al.* [47] built an architecture timing model for PowerPC processor. The approach focus is to apply workloads that enforce certain invariants by multiple executions of the same loops. For example, the IPC for executing a loop should not decrease when the buffer queue size increases. However, a different microbenchmark would need to be created for every different test, which constraints the coverage space, and although a test failure provides certain symptoms of a performance bug, it does not assert the presence of the bug to any specific location. Although this technique is useful, its effectiveness is restricted only to a few types of performance bugs.

The work by Singhal *et al.* [4] described the procedures and tools used for performance verification in the Intel Pentium 4 processor on a 90nm technology. This work shows that detection of performance bugs relies on the execution of benchmarks and checking whether the design achieved the expected performance, otherwise, a manual debug process would be needed. A similar approach is also applied for identifying I/O system performance anomaly in work presented by Shen *et al.* [48].

Overall, the model comparison-based approach has two main drawbacks. First, the same performance bug can appear in both the model and the design, as described by Ho [28], and consequently cannot be detected. Although some works strive to find an accurate golden reference [46],

such effort is restricted to special cases and is very difficult to generalize, in most cases these models make over-simplifications on the model, that would not work well on modern microprocessor designs.

In particular, in presence of intrinsic performance variability [49, 50], finding a golden reference for general cases becomes almost impossible. Second, constructing a reference architecture model is labor-intensive and time-consuming. On one hand, it is very difficult to build a general model applicable across different architectures. On the other hand, building one model for every new architecture is not cost-effective.

3.2.4 Microprocessor performance bug localization

The works on bug localization are even fewer [51, 52] than those in bug detection. The work by Utamaphetai *et al.* [51] models the entire design as buffers and finite state machines in order to use strategies developed for test generation, these simplifications make the methodology unable to localize bugs that are not necessarily related to state transitions or dependencies. The work by Adir *et al.* [52] performs a microarchitectural-level verification based on a test plan created by using coverage models. These coverage models, however, require "creativity" and significant domain knowledge. Since the analysis of test program results is still manual, their evaluation required almost 10 days (without considering the time to create the coverage models) to validate a single microarchitectural unit. Scalability concerns can arise when other units and the interactions between are considered. Both of these works focus on test generation techniques while the analysis of test results is still a conventional manual approach.

3.2.5 Performance bugs in other domains

A related performance issue in datacenter computing is performance anomaly detection [53]. The main techniques here include statistics-based, such as ANOVA tests and regression analysis, and machine learning-based classification. Although there are many computers in a datacenter, the subject is the overall system performance upon workloads in very coarse-grained metrics. As such, the normal system performance is much better defined than individual processors. Perfor-

mance bug detection is mentioned for distributed computing in clouds [54]. In this context, the overall computing runtime of a task is greater than the sum of runtimes of computing its individual components as extra time is needed for the data communication. However, the difference should be limited and otherwise, an anomaly is detected. As such, the performance debug in distributed computing is focused on communication and assumes that all processors are bug-free. Evidently, such an assumption does not hold for processor microarchitecture designs. In another work, cloud service anomaly prediction is realized through a self-organizing map, an unsupervised learning technique [55]. Gan *et al.* [56] developed a deep learning-based online QoS violation prediction technique for cloud computing by using runtime traces. This work detects an upcoming violation and identifies the microservice responsible for it. This is similar to the localization task tackled in this work, but in a different environment where the source corresponds to a microservice being executed by a system, as opposed to a microarchitectural unit that is part of the system itself.

Performance bug is also studied for software systems where bugs are detected by users or code reasoning [57]. A machine learning approach is developed for evaluating software performance degradation due to code change [58]. Software code analysis [59] is used to identify performance-critical regions when executing in cloud computing. Parallel program performance diagnosis is studied by Atachiants *et al.* [60]. Performance bugs in smartphone applications are categorized into a set of patterns [61] for future identification. In other works, a common strategy for performance bug localization, is to use a language processing model based on bug reports from software project databases [62, 63, 64, 65]. The models learn to identify the correlations between specific keywords in bug reports and the code changes implemented to fix them. As the degree of concurrency in microprocessor architectures is usually higher than a software program, performance debugging for microprocessors is generally much more complicated.

3.3 Machine learning for performance debugging

Through this chapter multiple methodologies that address two of the main tasks required for a proper performance debugging are introduced. The first one is **bug detection**, which is discussed in Section 3.4 and consists in determining whether a new architecture has performance bugs or not.

Note that simply detecting a bug in the pre-silicon phase or early in the post-silicon phase is highly valuable to shorten the validation process or to avoid costly steppings. Further, bug detection is a challenge by itself. The second task is **bug localization**, which is an even more challenging problem, since the goal is to provide designers with information regarding where in the design that bug is present. This task is discussed in Section 3.5.

The methodologies for detection and localization of performance bugs proposed in this work are based on machine learning methods, as they have been proven to be extraordinary on extracting knowledge from data, and handling very complicated non-linear behaviors. These characteristics make ML the best approach to mimic a human manual process among other mathematical or algorithmic options.

As common ground for all the machine learning methodologies presented in this dissertation, we rely on the usage of **performance probes**, which are used to evaluate the design and assess the obtained performance. A detailed description of what these probes are, how they are extracted, and why these are useful is presented in Section 3.3.1.

3.3.1 Performance probe design

The term "performance probe" is defined in this work as a tuple composed by a microbenchmark and a corresponding set of performance counters that will be monitored in order to evaluate the design performance. Sections 3.3.1.1 and 3.3.1.2 elaborate on the methodology followed to design these.

3.3.1.1 Microbenchmark extraction

Finding the right applications to use as "probes" for microarchitectural performance bugs is critical to the efficiency and efficacy of the proposed methodologies. One possible approach could be to meticulously hand generate an exhaustive set of microbenchmarks to test all the interacting components of the microarchitecture under all possible instruction orderings, branch paths, dependencies, etc. This approach has been attempted before by prior works [66, 67, 68].

While this approach would be possible for small and simple designs, it is unrealistic for mod-

ern microprocessor cores and therefore, automating the process as much as possible is highly desirable, given the overheads already required for verification and validation. Thus, another key goal of this work is to automatically find and select short, orthogonal, and performance-relevant microbenchmarks to use in the microarchitecture probing.

Typically in computer architecture research, large, orthogonal suites of workloads, such as the SPEC CPU suites [32, 69], are used for performance benchmarking. The applications in these suites, however, might take weeks or even months to simulate to completion, making their usage unfeasible. Thus, the development of statistically accurate means to shorten benchmark runtimes has received significant attention [70, 71, 72]. One notable work was SimPoints [73, 72, 74, 75], wherein the authors propose a methodology to identify short, performance-orthogonal segments of long-running applications. These segments are simulated separately and then the whole application performance is estimated as a weighted average of the performance of those segments. In this work, a novel use of the SimPoints methodology is proposed. Instead of using the SimPoints as originally intended, to estimate the performance of the larger application, they are used to *automatically* identify and extract short, orthogonal, and performance-relevant microbenchmarks from long-running applications. This work leverages SimPoints' identification of orthogonal basic-block vectors in the given program to be the source of orthogonal microbenchmarks.

As an example of how this provides greater visibility into performance bugs, consider Figure 3.2. In the figure, a comparison of the performance per SimPoint of the bug-free and *Bug 1* versions of Skylake (described in Section 3.1) for the benchmark 403.gcc in SPEC CPU2006 [32] is shown. Although the overall difference in the whole application (leftmost bars) is less than 1% , once the performance is split by SimPoint it can be observed that SimPoint #12 has a degradation of over 20%, making *Bug 1* much easier to detect.

The reason why SimPoint #12 shows this behavior is because it has more than twice the number of *xor* operations than the others, accounting for 2.3% of all instructions executed in that SimPoint. Even though the overall impact on the performance in 403.gcc is very low, this particular SimPoint represents well a class of applications with similar behavior that would not be represented by

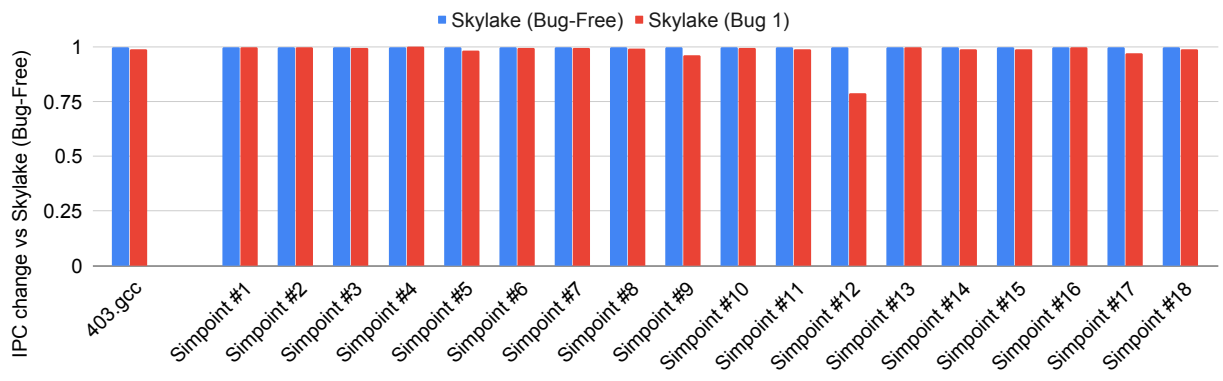


Figure 3.2: IPC by SimPoints in 403.gcc for Skylake architecture.

looking at any single benchmark in the SPEC CPU suite. Thus, by utilizing SimPoints individually more performance bug coverage can be gained than by looking at full application performance.

3.3.1.2 Performance counters as ML features

Performance counter data from microbenchmark simulation will constitute the main features for the machine learning models used for this work. Hence, they are essential components of the performance probes. There are two main reasons why the performance counters are used throughout this work. First, the general availability of performance counters in almost any architecture avoids the overhead of dedicated data acquisition infrastructure. Second, an architecture usually has hundreds or thousands of performance counters, which are generally sufficient to extract necessary information for performance estimation at microarchitecture level. It should be noted that, as part of the design cycle, performance counters often undergo explicit validation. Thus, the assumption of their sanity is central for performance debugging in general, not only this methodology.

Using the entire set of available performance counters as features would unnecessarily extend training and inference time for machine learning models, and even degrade them as some useless counters may introduce noise into the models. Hence, a small subset of them is selected for each microbenchmark, these are sufficient for effective machine learning inference on that model. We note that the abundance of counters along with the selection method makes the methodologies resilient to small changes in the counters available across different generations of architecture

designs.

The counter selection is carried out in two steps, as described below.

- **Step 1:** The Pearson correlation coefficient is evaluated for each counter with respect to a performance metric, such as IPC. Only those counters with high correlations with IPC (magnitude greater than a threshold α) are retained and all the others are pruned out.

- **Step 2:** Among the remaining counters, the correlation between every pair of them is evaluated. If two counters have a Pearson correlation coefficient whose magnitude is greater than a threshold β, they are considered redundant and the one with the lowest correlation with IPC is removed.

Note that counter selection is conducted independently for each probe and thus different probes may use different counters. Examples of the most commonly selected counters are the number of fetched instructions per cycle, the percentage of branch instructions, the number of writes to registers, the percentage of correctly predicted indirect branches, the number of cycles on which the maximum possible number of instructions were committed, among others.

3.4 Machine learning for performance bug detection

3.4.1 Problem formulation for the bug detection task

The work presented in this section has the goal to detect the existence of performance bugs given a new microprocessor design. The bug detection task can be formulated as a binary classification task for which positive cases correspond to architecture designs with performance bugs, and negative cases correspond to bug-free architectures.

The focus is placed specifically on microarchitectural-level design bugs, and circuit-level timing bugs that affect the design in terms of clock period are considered out of the scope of this work. The availability of one or more presumably bug-free legacy microarchitectural designs is assumed. Given the extensive pre-silicon and post-silicon debug, this assumption generally holds. There are not any assumptions regarding how a bug may look like or behave.

3.4.2 Methodology overview for the bug detection task

Performance bug detection bears similarities to standard chip functional bug detection. Thus, a testbench strategy similar to the one frequently used for functional verification is sensible. In other words, a set of microbenchmarks are simulated on a new architecture and certain performance characteristics are monitored and analyzed. The key difference is that, in functional verification, designers have access to the correct responses to compare against, while the bug-free performance for a new microarchitecture is not well defined.

The proposed methodologies for bug detection rely on the usage performance counter data as input, as these counters are readily available on most designs. The counter data is extracted via simulation (or execution) of a diverse set of microbenchmarks. For each microbenchmark, a specialized set of counters that is highly associated with its performance behavior is selected, and together, they constitute a **performance probe**, which were described in Section 3.3.1. In Section 3.4.3 a naïve baseline approach to solving the bug detection problem is introduced. This method serves as a prelude for describing the proposed methodology in Section 3.4.4. In addition, it provides a reference for comparison in the experimentation.

3.4.3 Baseline single-stage naïve bug detection approach

This approach makes use of supervised machine learning to classify whether a microarchitecture has performance bugs or not. This methodology takes as input features performance counter data, a target performance metric, such as the IPC (number of Instructions committed Per Cycle), as well as microarchitecture design parameters such clock period, cache sizes, among others. One classification result on bug existence (or lack thereof) is generated for each application. The overall detection result is then obtained by a voting ensemble, based on the classification results of multiple applications.

Further, as it was explained in Section 3.3.1, simulating an application to completion would require a huge runtime (in the order of weeks or months). So, in this case, we use the data from all the SimPoints in one application as inputs for the bug classification. Figure 3.3 shows how

a classifier for one application with 'N' SimPoints would look. Multiple instances of this are generated, one per application in the microbenchmark set.

One classification is performed per application, after that, a voting mechanism ensembles the results to produce a final classification. Let ρ be the ratio of the number of applications with positive classification (indicating a bug) versus the total number of applications. A design is then classified as buggy if $\rho \geq \theta$, where θ is a threshold determining the tradeoff between true positives and false positives. The designs used for the training of this model consists of legacy designs, both with and without bugs. The bugs we use for this purpose were artificially and purposely injected in order to obtain the labeled data.

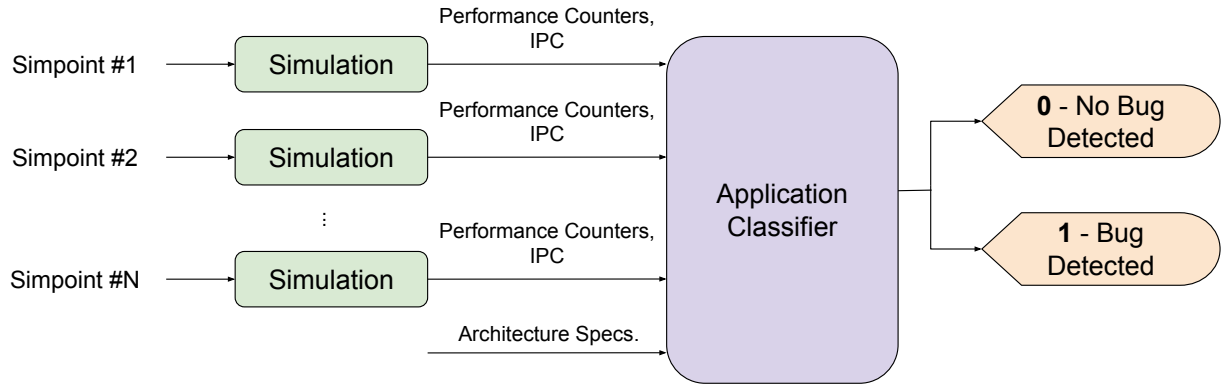


Figure 3.3: Overview of a naïve approach for bug detection. Multiple instances of this classifier are implemented, one per application, and the final result is produced by a voting mechanism.

3.4.4 Two-stage bug detection approach

The proposed methodology is composed of two stages. The first stage consists of a machine learning model trained with performance counter data from (presumably) bug-free microarchitectures, its goal is to infer microprocessor performance, for this work the models are configured to perform inference of IPC, but other performance metrics might be used, such as CPI or AMAT, depending on the needs of the specific setup. If such a model is well-trained and achieves high accuracy, it will capture the relationships between the performance counters and IPC in the case

of bug-free microarchitectures. When this model is then applied to a design with bugs, a significant increase in inference errors is expected, as the bugs ruin the relations learned by the model between the counters and IPC. As such, the second stage determines bug existence using the IPC inference errors obtained in the first stage. An overview of the two-stage methodology is depicted in Figure 3.4.

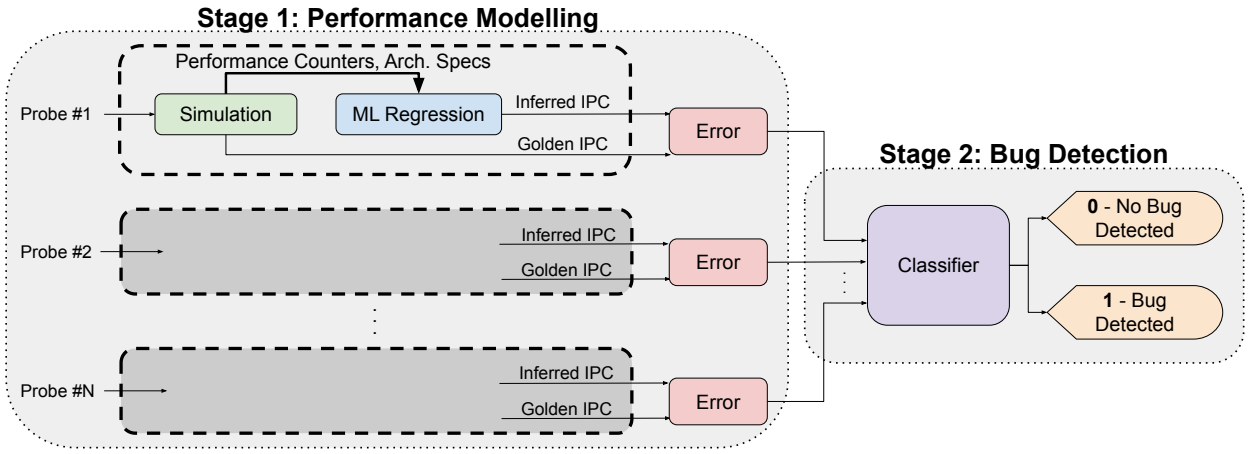


Figure 3.4: Overview of the two-stage methodology for bug detection.

Although the methodology is evaluated using only microarchitecture simulations, its key idea can be easily extended for FPGA prototyping or post-silicon processor chip testing using built-in performance counters.

Sections 3.4.4.1 and 3.4.4.2 elaborate on the details of each of the two stages composing this methodology.

3.4.4.1 Stage 1: Performance inference

Stage 1 of the methodology models overall bug-free processor performance (for this work IPC is used, but the methodology can be used with other metrics as well), via machine learning regression models. The model training data are taken only from bug-free (at least presumably) designs. Otherwise, the inference errors between bug-free designs and bug cases cannot be differentiated.

One model is trained for each probe. As workload characteristics of different probes can be drastically different, it is very difficult (and also unnecessary), for a single model to capture them all. A separate model for each probe is used to achieve high accuracy for bug-free designs. This represents a key difference from previous methods of counter-based performance predictions where a single generic model is derived so that it works across many different workloads [42, 43, 44, 45]. By tuning the models to the particular counters relevant to each probe workload, a much higher accuracy can be achieved.

Every model is trained using a variety of different microarchitectures. This is to ensure that inference error differences are due to the existence of performance bugs, and not merely microarchitecture differences between the legacy and the new designs. In practice, the training is performed on legacy designs and the model inference would be on a new microarchitecture.

For this methodology, the input features are taken as a time series, as opposed to the single-stage methodology described in Section 3.4.3, where the counters were represented by a single value for the entire SimPoint. The simulation (or running system) proceeds over a series of time steps $t_1, t_2, ...$ Let the current time step be t_i, then the model takes feature data of $t_i, t_{i-1}, ..., t_{i-W}$ as input to estimate the IPC at time step t_i, where W represents the history window size. Usually, a time step consists of several clock cycles, e.g., a half-million cycles, and after every sample, every counter is reset, so their values reflect the only the behavior of the current time step and do not carry any history information. If the intention is to take the aggregated data of an entire SimPoint as input, that is a special case of this framework, where the step size is the entire SimPoint and the window size is 1.

Besides the selected counters discussed in Section 3.3.1.2, some microarchitecture design parameters are used as features for the machine learning models. They include clock period, pipeline width, reorder buffer size, and some cache characteristics such as latency, size, and associativity. Please note that the parameter features are constant in the time series input for any specific microarchitecture.

3.4.4.2 Stage 2: Bug detection classifier

Stage 2 of this methodology uses a vector of IPC inference error metrics $\vec{\Delta}$ obtained from the results of stage 1. The purpose of this stage is to identify if a performance bug exists in a new microarchitecture design using those errors. In general, small error values indicate that the machine learning model from stage 1, which is trained with bug-free designs, matches well the design under test and hence likely no bugs are present in the design under test. On the other hand, large error values indicate a high likelihood of performance bugs being present in the design. Section 3.4.4.2.1 describes all the error metrics evaluated for this work.

3.4.4.2.1 *Evaluated error metrics*

Once the IPC inference is performed, multiple error metrics between the ML model estimation and the data obtained via simulation are calculated. This error data, or at least a subset of it, constitutes the input for the second stage of the methodology.

Consider the set of probes $P = \{p_1, p_2, ...\}$. A trained model is applied to infer IPC $\hat{y}_{i,j}$ of probe p_i at time step t_j for a particular microarchitecture, with T_i being the total number of time steps in probe p_i. The corresponding IPC obtained by simulation is denoted as $y_{i,j}$. Several error metrics were evaluated, these are as follows:

(a) The Mean Absolute Error, defined as

$$\text{MAE} = \frac{1}{T_i} \sum_{j=1}^{T_i} |\hat{y}_{i,j} - y_{i,j}| \tag{3.1}$$

(b) The Relative Mean Absolute Error, defined as

$$\text{RMAE} = \frac{\sum_{j=1}^{T_i} |\hat{y}_{i,j} - y_{i,j}|}{\sum_{j=1}^{T_i} y_{i,j}} = \frac{\text{MAE}}{\frac{1}{T_i} \sum_{j=1}^{T_i} y_{i,j}} \tag{3.2}$$

which corresponds to the Mean Absolute Error divided by the average value of the simulated IPC across all the T_p time steps.

(c) The Mean Squared Error, defined as

$$\text{MSE} = \frac{1}{T_i}\sum_{j=1}^{T_i}(\hat{y}_{i,j} - y_{i,j})^2 \tag{3.3}$$

(d) The Root Mean Squared Error, defined as

$$\text{RMSE} = \sqrt{\frac{1}{T_i}\sum_{j=1}^{T_i}(\hat{y}_{i,j} - y_{i,j})^2} = \sqrt{\text{MSE}} \tag{3.4}$$

which corresponds to the square root of the Mean Squared Error.

(e) The Relative Root Mean Squared Error, defined as

$$\text{RRMSE} = \frac{\sqrt{\frac{1}{T_i}\sum_{j=1}^{T_i}(\hat{y}_{i,j} - y_{i,j})^2}}{\frac{1}{T_i}\sum_{j=1}^{T_i} y_{i,j}} = \frac{\text{RMSE}}{\frac{1}{T_i}\sum_{j=1}^{T_i} y_{i,j}} \tag{3.5}$$

which corresponds to the Root Mean Squared Error divided by the average value of the simulated IPC across all the T_p time steps.

(f) The maximum prediction error across the whole trace, defined as:

$$\text{Max} = \max_{1 \leq t_j \leq T_i}(\hat{y}_{i,j} - y_{i,j}) \tag{3.6}$$

(g) The largest over-prediction across the whole trace, in most cases this will be the same as the maximum prediction error, except that it saturates at 0. In other words, if there is no time step at which $\hat{y}_{i,j} \geq y_{i,j}$ then this metric is zero, it is the same as the maximum prediction error otherwise. Mathematically it can be defined as:

$$\text{Over-Prediction} = \max\left(\max_{1 \leq t_j \leq T_i}(\hat{y}_{i,j} - y_{i,j}), 0\right) \tag{3.7}$$

(h) The minimum prediction error across the whole trace, defined as:

$$\text{Min} = \min_{1 \leq t_j \leq T_i} (\hat{y}_{i,j} - y_{i,j}) \tag{3.8}$$

(i) The largest under-prediction across the whole trace, in most cases this will be the same as the minimum prediction error (largest negative error), except that it saturates at 0. In other words, if there is no time step at which $\hat{y}_{i,j} \leq y_{i,j}$ then this metric is zero, it is the same as the minimum prediction error otherwise. Mathematically it can be defined as:

$$\text{Under-Prediction} = \min \left(\min_{1 \leq t_j \leq T_i} (\hat{y}_{i,j} - y_{i,j}), 0 \right) \tag{3.9}$$

(j) The Area under the error curve, this corresponds to the area of difference between the inferred IPC over time and the actual (simulated) IPC. It is also approximately equal to the total absolute error, it is defined as:

$$\text{AUC} = \frac{1}{2} \sum_{j=2}^{T_i} \left(|\hat{y}_{i,j} - y_{i,j}| + |\hat{y}_{i,j-1} - y_{i,j-1}| \right) \tag{3.10}$$

(k) The Area under positive error curve, it corresponds only to the segments of the error trace where the error is positive, that means $\hat{y}_{i,j} \geq y_{i,j}$. It can be defined as:

$$\text{Positive AUC} = \frac{1}{2} \sum_{j=2}^{T_i} \left(\max \left[(\hat{y}_{i,j} - y_{i,j}), 0 \right] + \max \left[(\hat{y}_{i,j-1} - y_{i,j-1}), 0 \right] \right) \tag{3.11}$$

(l) The Area under negative error curve, it corresponds only to the segments of the error trace where the error is negative, that means $\hat{y}_{i,j} \leq y_{i,j}$. It can be defined as:

$$\text{Negative AUC} = \frac{1}{2} \sum_{j=2}^{T_i} \left(\min \left[(\hat{y}_{i,j} - y_{i,j}), 0 \right] + \min \left[(\hat{y}_{i,j-1} - y_{i,j-1}), 0 \right] \right) \tag{3.12}$$

(m) The Pearson Correlation factor between the simulated IPC and the infered IPC. This can be

defined as:

$$\text{Pearson Correlation} = \frac{T_i \sum_{j=1}^{T_i} y_{i,j}\hat{y}_{i,j} - \sum_{j=1}^{T_i} y_{i,j} \sum_{j=1}^{T_i} \hat{y}_{i,j}}{\sqrt{T_i \sum_{j=1}^{T_i} y_{i,j}^2 - \left(\sum_{j=1}^{T_i} y_{i,j}\right)^2}\sqrt{T_i \sum_{j=1}^{T_i} \hat{y}_{i,j}^2 - \left(\sum_{j=1}^{T_i} \hat{y}_{i,j}\right)^2}} \tag{3.13}$$

For each probe $p_i \in P$ let $\vec{\delta_i}$ be the vector of error metrics which contains all, or a subset of these metrics. Then, $\vec{\Delta} = [\vec{\delta_1}, \vec{\delta_2}, ...\vec{\delta_{|P|}}]$. The vector $\vec{\Delta}$ is fed to stage 2 of the methodology as input, so that it can identify if a bug exists in the corresponding microarchitecture.

Since bugs can reside at different locations and manifest in different ways, multiple orthogonal probes are necessary to improve the breadth and robustness of the detection mechanism.

Two methodologies for the implementation of this second stage will be presented, discussed, and contrasted with detail in upcoming sections. In Section 3.4.4.2.2 a heuristic rule-based methodology is described, and in Section 3.4.4.2.3 a machine learning-based approach is proposed.

3.4.4.2.2 *Rule-based classifier*

For this method only one of the error metrics described in Section 3.4.4.2.1 is used. In other words, for probe p_i the vector of error metrics $\vec{\delta_i}$, is composed of a single element, therefore, for the remaining of this section we refer to this error as δ_i. Each $\vec{\Delta}$, which contains the selected error metric from all probes in P is considered a single sample. This means that a single sample is obtained for each available microarchitecture. This data arrangement makes the number of available data samples very scarce, which makes this task difficult to solve by applying machine learning techniques. For this reason, a heuristic rule-based classifier is proposed.

Suppose there are m positive samples (architectures with bugs) and n negative samples (bug-free architectures) in the training set. For each probe $p_i \in P$, the mean μ_i^+ and standard deviation σ_i^+ of the IPC inference error metric among the m positive samples is computed. Similarly, μ_i^- and σ_i^- among the n negative samples for each $p_i \in P$. These statistics constitute a reference that can be used to evaluate the IPC inference errors for new architectures.

For the error δ_i' of applying probe p_i on the new architecture, the following ratios based on the

statistics of the labeled data are evaluated.

$$\gamma_i^+ = \frac{\delta_i'}{\mu_i^+ + \omega\sigma_i^+} \ , \ \ \gamma_i^- = \frac{\delta_i'}{\mu_i^- + \omega\sigma_i^-} \tag{3.14}$$

where ω is a parameter selected using the labeled data. In general, a large value of δ_i' signals a high probability of bugs in the new microarchitecture. Ratios γ_i^+ and γ_i^- are relative to the labeled data and make the errors from different probes comparable.

Therefore, for each new sample the vectors of scores $[\gamma_1^+, \gamma_2^+, ..., \gamma_{|P|}^+]$ and $[\gamma_1^-, \gamma_2^-, ..., \gamma_{|P|}^-]$ are obtained, then, our classifier is a rule-based recipe as follows.

1. If $\max(\gamma_1^+, \gamma_2^+, ..., \gamma_{|P|}^+) > \eta$, where η is a parameter, this architecture is classified to have bugs. This is for the case where a large error appears in at least one probe regardless of errors from the other probes.

2. If $(\gamma_1^- + \gamma_2^- + ... + \gamma_{|P|}^-)/|P| > \lambda$, where $\lambda < \eta$ is a parameter, this architecture is classified to have bugs. This is for the case where small errors appear on many probes.

3. For other cases, the architecture is classified as bug-free.

While the values of η and λ are empirically chosen, the value of ω is trained according to the true positive rate and false positive rate on the labeled data. In the training, a set of uniformly spaced values in a range are tested for ω, and the one with the maximum true positive rate and false positive rate no greater than 0.25 is chosen.

3.4.4.2.3 *Machine learning-based classifier*

In this classifier approach, a classification is performed for every probe available. Using every probe as a different sample, provides many more data points, making it possible to use a machine learning approach. Since every design will end up with $|P|$ predictions, an ensemble method is needed to determine the outcome for the overall design.

The used approach is similar to the one implemented in Section 3.4.3. In this methodology, if the number of probes classified as buggy is higher than a threshold, the design is classified as

buggy. The threshold is defined by using the legacy designs in the training set. A threshold is picked such that it maximizes the number of bugs detected while keeping the number of false alarms below a particular value.

3.4.5 Experimental setup for bug detection

This section elaborates on the details of the implementation of the proposed methodologies. Section 3.4.5.1 covers the setup of the probes utilized in order to perform the performance bug detection, Section 3.4.5.2 covers the implementation of different microarchitectures used to evaluate the methodology and Section 3.4.5.3 enlists the bugs that were injected to assess the quality of the methodologies.

3.4.5.1 Probe setup

The performance probes used for this evaluation are extracted from 10 applications in the SPEC CPU2006 benchmark [32] using the SimPoints methodology [75], as described in Section 3.3.1.1. The methodology is evaluated using SPEC CPU2006, instead of the more recent SPEC CPU2017 [69] suite, because of their relatively smaller memory footprints, greater compatibility with the gem5 [33] simulation environment, and generally reduced running times and computational resources needed to detect performance bugs. Despite of this, there is no reason why these techniques would not work with SPEC CPU2017 applications or any other benchmark suite. Unlike the case when performance improvement techniques are being developed, and therefore the benchmarks serve as test cases that should be exhaustively used, the selected applications here are part of the methodology implementation. In fact, the selection (which and how many) affects the tradeoff between the efficacy of detection and the runtime overhead of the methodology, as will be shown in later sections.

In total 190 SimPoints were extracted from the ten SPEC CPU2006 applications used. Each of these SimPoints contains around 10M instructions and their performance counters are sampled every 500k cycles. Every time the counters are sampled, they are reset, so their value is not cumulative. The selected applications correspond to an arbitrary set of the benchmark that were

able to compile and run successfully across all microarchitectures. Adding more benchmarks would improve the achieved results as it increases the coverage of behaviors evaluated. A list of the applications, the types of operands it uses, the programming language on which they are written, their usage purpose and the number of extracted SimPoints is provided in Table 3.1.

Table 3.1: Applications from SPEC CPU2006 benchmark selected for usage in evaluation of performance debug techniques.

Benchmark	Operand Type	Language	Application	# SimPoints
400.perlbench	Integer	C	PERL Programming Language	14
401.bzip2	Integer	C	Compression	23
403.gcc	Integer	C	C Compiler	18
426.mcf	Integer	C	Combinatorial Optimization	15
433.milc	Floating Point	C	Quantum Chromodynamics	20
436.cactusADM	Floating Point	C/Fortran	General Relativity	16
444.namd	Floating Point	C++	Molecular Dynamics	26
450.soplex	Floating Point	C++	Linear Programming	21
458.sjeng	Integer	C	AI: Chess	19
462.libquantum	Integer	C	Quantum Computing	18

3.4.5.2 Simulated architectures

All experiments are performed via gem5 [33] simulations in System Emulation mode. Note that the key ideas of the methodology are not constrained to the evaluated setup, and can be implemented in other simulators, as well as post-silicon debug. Here, the focus is on core performance bugs, thus the Out-Of-Order core (O3CPU) model with the x86 ISA is used. The gem5 simulator is configured to model the behavior of eight different existing microarchitectures: Intel Broadwell, Cedarview[2], Ivy Bridge, Skylake, and Silvermont, as well as AMD Jaguar, K8, and K10. Additionally, 12 artificial microarchitectures with realistic settings are created by varying clock period, reorder buffer size, cache size, latency, associativity and number of levels, variations on functional unit quantity, latency, and port organization, as well as other configuration variables. These microarchitectures serve as training data for machine learning models and unseen data for testing the models. Core specifications of these 20 microarchitectures are summarized in Table 3.2 and cache

[2]Implements a Cedarview-like superscalar architecture but assumes out-of-order completion

characteristics are shown in Table 3.3. Table 3.4 describes the port organization for each of the architectures.

Table 3.2: Core specifications of the architectures used to evaluate the bug detection mechanisms.

Architecture	**Clock Cycle**	**CPU Width**	**ROB Size**	**Func. Unit Latency (FP / Multiplier / Divider)**
Broadwell	4.0GHz	4	192	5 cycles / 3 cycles / 20 cycles
Cedarview	1.8GHz	2	32	5 cycles/ 4 cycles / 30 cycles
Ivy Bridge	3.4GHz	4	168	5 cycles / 3 cycles / 20 cycles
Jaguar	1.8GHz	2	56	4 cycles / 3 cycles / 20 cycles
K8	2.0GHz	3	24	4 cycles / 3 cycles / 11 cycles
K10	2.8GHz	3	24	4 cycles / 3 cycles / 11 cycles
Silvermont	2.2GHz	2	32	2 cycles / 7 cycles / 69 cycles
Skylake	4.0GHz	4	256	4 cycles / 4 cycles / 20 cycles
Artificial 0	2.5GHz	4	192	5 cycles / 3 cycles / 20 cycles
Artificial 1	1.5GHz	4	192	4 cycles / 3 cycles / 11 cycles
Artificial 2	4.0GHz	8	168	4 cycles / 4 cycles / 20 cycles
Artificial 3	3.0GHz	8	32	4 cycles / 4 cycles / 20 cycles
Artificial 4	4.0GHz	2	192	5 cycles / 3 cycles / 20 cycles
Artificial 5	3.5GHz	2	32	4 cycles / 3 cycles / 11 cycles
Artificial 6	3.5GHz	4	192	4 cycles / 4 cycles / 20 cycles
Artificial 7	3.0GHz	4	32	2 cycles / 7 cycles / 69 cycles
Artificial 8	3.0GHz	2	192	4 cycles / 3 cycles / 11 cycles
Artificial 9	3.5GHz	8	192	4 cycles / 3 cycles / 11 cycles
Artificial 10	1.5GHz	8	32	5 cycles/ 4 cycles / 30 cycles
Artificial 11	3.5GHz	4	32	5 cycles/ 4 cycles / 30 cycles

Table 3.3: Cache specifications of the architectures used to evaluate the bug detection mechanisms.

Architecture	**L1 Cache (Size / Assoc. / Latency)**	**L2 Cache (Size / Assoc. / Latency)**	**L3 Cache (Size / Assoc. / Latency)**
Broadwell	32kB / 8-way / 4 cycles	256kB / 8-way / 12 cycles	64MB / 16-way / 59 cycles
Cedarview	32kB / 8-way / 3 cycles	512kB / 8-way / 15 cycles	No L3
Ivy Bridge	32kB / 8-way / 4 cycles	256kB / 8-way / 11 cycles	16MB / 16-way / 28 cycles
Jaguar	32kB / 8-way / 3 cycles	2MB / 16-way / 26 cycles	No L3
K8	64kB / 2-way / 4 cycles	512kB / 16-way / 12 cycles	No L3
K10	64kB / 2-way / 4 cycles	512kB / 16-way / 12 cycles	6MB / 16-way / 40 cycles
Silvermont	32kB / 8-way / 3 cycles	1MB / 16-way / 14 cycles	No L3
Skylake	32kB / 8-way / 4 cycles	256kB / 4-way / 12 cycles	8MB / 16-way / 34 cycles
Artificial 0	64kB / 2-way / 4 cycles	512kB / 4-way / 12 cycles	No L3
Artificial 1	64kB / 8-way / 5 cycles	2MB / 8-way / 16 cycles	No L3
Artificial 2	32kB / 2-way / 5 cycles	256kB / 8-way / 16 cycles	No L3
Artificial 3	32kB / 2-way / 3 cycles	512kB / 16-way / 24 cycles	8MB / 32-way / 52 cycles
Artificial 4	64kB / 8-way / 3 cycles	1MB / 8-way / 20 cycles	32MB / 16-way / 28 cycles
Artificial 5	32kB / 4-way / 5 cycles	256kB / 4-way / 16 cycles	8MB / 32-way / 44 cycles
Artificial 6	64kB / 8-way / 4 cycles	1MB / 8-way / 16 cycles	8MB / 32-way / 36 cycles
Artificial 7	16kB / 8-way / 3 cycles	512kB / 16-way / 12 cycles	32MB / 32-way / 28 cycles
Artificial 8	32kB / 2-way / 2 cycles	1MB / 16-way / 16 cycles	32MB / 32-way / 52 cycles
Artificial 9	16kB / 4-way / 5 cycles	1MB / 4-way / 20 cycles	64MB / 16-way / 44 cycles
Artificial 10	32kB / 2-way / 2 cycles	256kB / 16-way / 24 cycles	64MB / 32-way / 36 cycles
Artificial 11	64kB / 4-way / 5 cycles	256kB / 4-way / 24 cycles	No L3

Table 3.4: Port organization of the architectures used to evaluate the bug detection mechanisms.

Set	Architecture	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6
I	Broadwell	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit
I	Cedarview	1 ALU, 1 Load Unit, 1 Store Unit, 1 Vector Unit, 1 Int Mult, 1 Divider	1 ALU, 1 Vector Unit, 1 FP Unit, 1 Branch Unit	1 Load Unit	1 Store Unit	-	-	-
I	Artificial 3	1 ALU, 1 Vector Unit, 1 FP Unit, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 FP Unit, 1 Int Mult	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit

Table 3.4: Port organization of the architectures used to evaluate the bug detection mechanisms.

Set	Architecture	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6
I	Artificial 4	1 ALU, 1 Vector Unit, 1 FP Unit, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 FP Unit, 1 Int Mult	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit
I	Artificial 5	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit
I	Artificial 7	1 ALU, 1 Vector Unit, 1 FP Unit, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 FP Unit, 1 Int Mult	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit
I	Artificial 8	1 Load Unit, 1 Store Unit	1 ALU, 1 Int Mult	1 ALU, 1 Branch Unit	1 FP Mult, 1 Divider	1 FP Unit	-	-

Table 3.4: Port organization of the architectures used to evaluate the bug detection mechanisms.

Set	Architecture	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6
I	Artificial 11	1 ALU, 1 Load Unit, 1 Store Unit, 1 Vector Unit, 1 Int Mult, 1 Divider	1 ALU, 1 Vector Unit, 1 FP Unit, 1 Branch Unit	1 Load Unit	1 Store Unit	-	-	-
I	Artificial 12	1 ALU, 1 Load Unit, 1 Store Unit, 1 Vector Unit, 1 Int Mult, 1 Divider	1 ALU, 1 Vector Unit, 1 FP Unit, 1 Branch Unit	1 Load Unit	1 Store Unit	-	-	-
II	Ivy Bridge	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Divider	1 ALU, 1 Vector Unit, 1 Int Mult, 1 FP Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit, 1 Branch Unit, 1 FP Unit	-

Table 3.4: Port organization of the architectures used to evaluate the bug detection mechanisms.

Set	Architecture	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6
II	Artificial 1	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit
II	Artificial 10	1 ALU, 1 Vector Unit, 1 Int Mult	1 ALU, 1 Vector Unit	1 ALU, 1 Vector Unit	1 Load Unit	1 Store Unit	1 FP Unit	1 FP Unit, 1 Branch Unit
III	Jaguar	1 ALU, 1 Vector Unit	1 ALU, 1 Vector Unit	1 FP Unit, 1 Int Mult	1 FP Mult, 1 Divider	1 Load Unit	1 Store Unit	-
III	Artificial 2	1 ALU, 1 Vector Unit, 1 Int Mult	1 ALU, 1 Vector Unit	1 ALU, 1 Vector Unit	1 Load Unit	1 Store Unit	1 FP Unit	1 FP Unit, 1 Branch Unit
III	Artificial 6	1 ALU, 1 Vector Unit, 1 Int Mult	1 ALU, 1 Vector Unit	1 ALU, 1 Vector Unit	1 Load Unit	1 Store Unit	1 FP Unit	1 FP Unit, 1 Branch Unit
III	Artificial 9	1 ALU, 1 Vector Unit, 1 Int Mult	1 ALU, 1 Vector Unit	1 ALU, 1 Vector Unit	1 Load Unit	1 Store Unit	1 FP Unit	1 FP Unit, 1 Branch Unit

Table 3.4: Port organization of the architectures used to evaluate the bug detection mechanisms.

Set	Architecture	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6
IV	K8	1 ALU, 1 Vector Unit, 1 Int Mult	1 ALU, 1 Vector Unit	1 ALU, 1 Vector Unit	1 Load Unit	1 Store Unit	1 FP Unit	1 FP Unit, 1 Branch Unit
IV	K10	1 ALU, 1 Vector Unit, 1 Int Mult	1 ALU, 1 Vector Unit	1 ALU, 1 Vector Unit	1 Load Unit	1 Store Unit	1 FP Unit	1 FP Unit, 1 Branch Unit
IV	Silvermont	1 Load Unit, 1 Store Unit	1 ALU, 1 Int Mult	1 ALU, 1 Branch Unit	1 FP Mult, 1 Divider	1 FP Unit	-	-
IV	Skylake	1 ALU, 1 Vector Unit, 1 FP Unit, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 FP Unit, 1 Int Mult	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit

For the purposes of training and testing the methodology these microarchitectures were partitioned into four disjoint sets, each with its own purpose on the different methodologies. A summary of this is shown in Table 3.5.

Table 3.5: Architecture partitioning for train / test of detection methodologies

	Set I	**Set II**	**Set III**	**Set IV**
# real architectures	2	1	1	4
# artificial architectures	7	2	3	0
Usage in Stage 1	Training	Early-stop validation	Not used	Testing
Usage in Stage 2	Not used	Training	Training	Testing
Usage in Single Stage Methodology	Training	Training	Training	Testing

Partitioning for the single stage methodology is much simpler than for the two-stage one, in the single stage methodology Set IV is used for testing, while every other architecture is used for training the models. A few observations should be made for the two-stage methodology:

- Architectures in Set I are used only for training the IPC estimation models. The reason to avoid using these architectures for training the second stage is because the inputs to stage 2 are error estimations. It is inevitable that architectures used for training in the first stage will obtain smaller errors than architectures that the models have not seen before, this might cause the models to flag an unseen bug-free architecture as buggy if the issue is not handled properly.

- To avoid the false positives mentioned earlier, a different set of architectures that were not used to train the first stage (sets II and III) are used to train the second stage. This provides an error reference of built with respect to the model behavior on unseen architectures.

- Set II is also used as a validation set for the first stage in the sense that training stops when no further improvement is obtained on this set. Despite being used for validation, the models are not specifically trained on these architectures.

- Set IV is used to evaluate all the methodologies, for this reason the set is composed of gem5 configurations that try to emulate the behavior of real architectures.

The assignment of each architecture to the specific set can be found on the leftmost column of Table 3.4

3.4.5.3 Implemented bugs

To achieve wide bug representation, we reviewed errata of commercial processors, consulted with industry experts, and tried to cover as many units as possible. Ultimately, fourteen basic types of bugs were developed and are summarized as follows.

1. Serialize X: Every instruction with the opcode of X is marked as a serialized instruction. This causes all future instructions (according to program order) to be stalled until the instruction with the bug has been issued.

2. Issue X only if oldest: Instructions whose opcode is X will only be retired from the instruction queue when it has become the oldest instruction there. This bug is similar to "POPCNT Instruction may take longer to execute than expected" found on Intel Xeon Processors [76]. Although this bug sounds similar to the "Serialize X", in this case, future instructions can be issued without any problem as long as they are not dependent on the instruction with the bug.

3. If X is oldest, issue only X: When an instruction whose opcode is X becomes the oldest in the instruction queue, only that instruction will be issued, even though other instructions might also be ready to be issued and the computation resources allow it.

4. If X depends on another instruction Y, delay T cycles. The commit of every instruction whose opcode is X and whose operands depend on the result of an instruction whose opcode is Y will be delayed by T clock cycles.

5. If less than N slots available in instruction queue, delay T cycles. If during a particular clock cycle less than N slots are available on the Instruction Queue, then the pipeline is stalled for

T cycles.

6. If less than N slots available in re-order buffer, delay T cycles. This bug is similar to the previous one, except that it is controlled by the number of available slots on the Re-Order Buffer, instead of the Instruction Queue.

7. If mispredicted branch, delay T cycles. When a branch misprediction occurs, the flushing of the pipeline is delayed by T cycles. This is done without affecting the correct program order.

8. If N stores to cache line, delay T cycles: After N stores have been executed to the same cache line, upcoming stores to the same line will be delayed by T cycles. This is a variation of "Store gathering/merging performance issue" found on the NXP MPC7448 RISC processor [77].

9. After N stores to the same register, delay T cycles. This bug is very similar to the previous, but instead of monitoring the stores to each cache line, it monitors the times each register has been written. After N times, every other instruction writing to that register will be delayed by T cycles. This bug is inspired by "GPMC may Stall After 256 Write Accesses in NAND_DATA, NAND_COMMAND, or NAND_ADDRESS Registers" found on TI AM3517, AM3505 ARM processor [78], with the difference that we generalized for any physical register, and in our case, the instruction is only stalled for a few cycles, as opposed to the actual bug, where the processor hanged. Another variation of this bug is implemented where the delay is applied once every N stores, instead of every store after the N-th.

10. L2 cache latency is increased by T cycles. This bug is inspired by the "L2 latency performance issue" on the NXP MPC7448 RISC processor [77].

11. Available registers reduced by N. The normal number of physical registers that are available to the microarchitecture is reduced by N.

12. If branch longer than N bytes, delay T cycles. If the jump that has to be executed after a

branch is longer than N bytes, then the pipeline is stalled T cycles longer than the time the processor required.

13. If X uses register R, delay T cycles. The commit of an instruction whose opcode is X and writes to the physical register index with N is delayed by T cycles. This bug is a variation of the "POPA/POPAD Instruction Malfunction" found on Intel 386 DX [79]. Due to that bug, the processor will hang if an instruction uses register EAX after a POP or POPAD instruction. In our version of the bug, we look for specific instructions using a specific physical register, if such a combination occurs, the instruction is delayed.

14. Branch predictor's table index function issue, reducing effective table size by N entries.

For each of these bug types, multiple variants are implemented by changing *X, Y, N, R*, and *T* respectively.

The bugs are grouped into four categories according to their average impact on the IPC obtained across the evaluated SPEC CPU2006 applications. A "High" impact means an average IPC degradation of 10% or greater. A "Medium" impact means the average IPC is degraded between 5% and 10%. A "Low" impact is between 1% and 5% and a "Very-Low" impact is less than 1%. Figure 3.5 shows the distribution of the severity of the average IPC impact for the implemented bugs.

3.4.5.4 Train and test data organization for the bug detection task

In the testing scheme, the goal is to detect performance bugs in new microarchitectures that were not included in the training data. Moreover, the utilized training data does not include any bug with the same type as the one in the testing data, i.e., the bug in the new microarchitecture is completely unseen in the training. In other words, the evaluation of each bug type is done individually, as different bug types must be left-out to evaluate each case. The data organization is as follows.

- Training data:

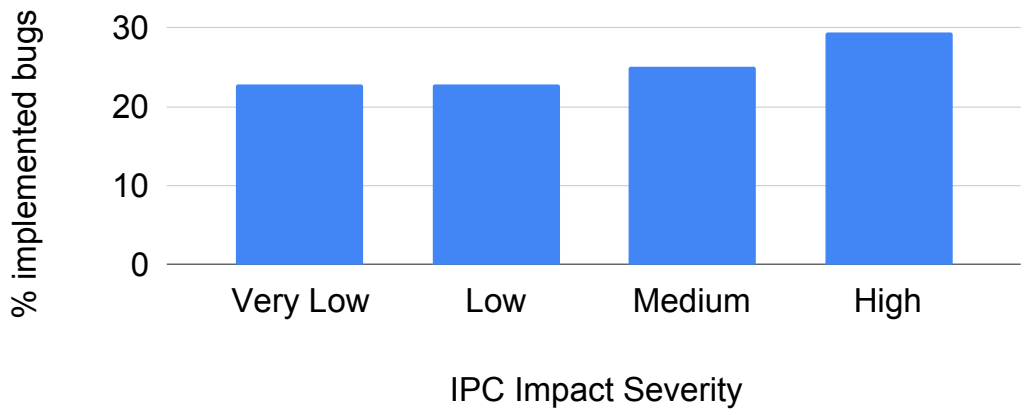


Figure 3.5: Average IPC impact distribution of bugs injected for performance bug detection.

1. Data with positive labels. IPC inference errors of all probes for microarchitectures in sets II and III with bug insertion. For each microarchitecture here, all types of bugs, except the one to be used in testing, are separately inserted. In each case, only one bug is inserted.
2. Data with negative labels. IPC inference errors of all probes on bug-free versions of the microarchitectures in sets II and III.

- Testing data:

1. Designs with bugs. Microarchitectures in set IV with all variants of a bug type inserted and this bug type is not included in training data. In each case, only one bug is inserted.
2. Bug-free versions of the microarchitectures in set IV.

An example, which has 3 bug types (1, 2, 3) and 2 variants for each type (*e.g.* Bug 1.1 and Bug 1.2 are two variants of the same bug type), is shown in Figure 3.6. Every time a new bug is evaluated, the model is trained again, using the corresponding data to match this scheme.

3.4.6 Bug detection results

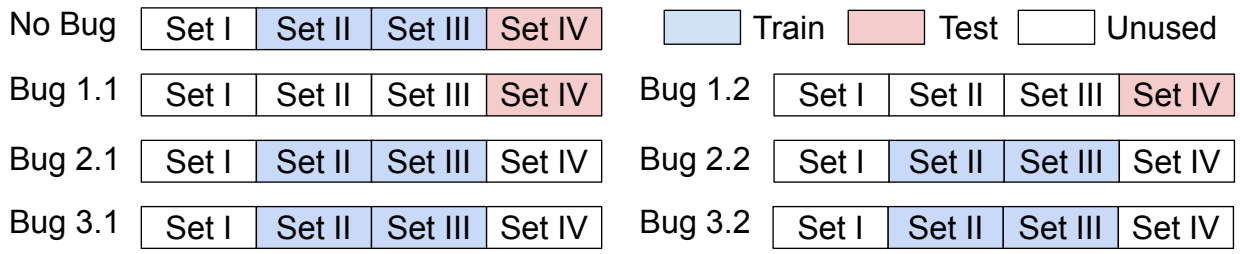


Figure 3.6: Example of the training and testing data split used for performance bug detection.

3.4.6.1 IPC estimation

In this section the results for the IPC modelling performed on the first stage of the proposed methodology are shown. The performance of several machine learning methods, such as Linear Regression, Neural Networks (in different flavors) and Gradient Boosted Trees were evaluated. For each of them different variations were studied.

The IPC estimation models for Multi-Layer Perceptrons [27], Convolutional Neural Networks [80] and Long Short-Term Memory networks [81] were implemented using the Python [22] library Keras [26]. For all these models, the Mean Squared Error (MSE) is used as loss function and Adam [82] is employed as the optimizer. A gradient clipping of 0.01 is enforced for LSTMs in order to avoid the gradient explosion issue, commonly seen when training recurrent networks. The gradient boosted trees are implemented via the XGBoost library [83] and the Lasso linear regression is implemented using Scikit-Learn [25].

The total training and inference runtime (considering all 190 probes), as well as inference error (in terms of area under error curve, as shown in Eq. (3.10)) for each IPC modelling technique are shown in Table 3.6. The name of each neural network-based method (LSTM, CNN and MLP), is prefixed with a number indicating the number of hidden layers and postfixed with the number of neurons in each hidden layer. The postfix number for a GBT method is the number of trees. Runtime results are measured on an Intel Xeon E5-2697A V4 processor with frequency of 2.6GHz and 512GB memory[3].

[3]This machine has no GPU, and GPU acceleration could significantly reduce these runtimes.

The total simulation time for detecting bugs in one new microarchitecture takes about 6 hours if the simulations are not executed in parallel.

Table 3.6: IPC modelling runtime and error statistics.

	Runtime		**Inference Error (Equation** (3.10)**)**			
ML Model	**Training**	**Inference**	**Average**	**Std. Dev.**	**Median**	**90th Perc.**
Lasso	0h 8m	3m 06s	10.2749	2.1598	10.3995	13.1740
SVM	0h 8m	3m 02s	5.2689	3.2091	4.6201	9.3797
1-LSTM-150	3h 22m	16m 21s	2.87×10^{7}	2.89×10^{8}	2.8082	5.4062
1-LSTM-250	4h 17m	17m 28s	4.94×10^{23}	6.71×10^{24}	2.8364	5.4582
1-LSTM-500	3h 39m	24m 18s	1.2×10^{6}	1.63×10^{7}	2.9193	4.7848
4-LSTM-150	4h 39m	54m 33s	5.81×10^{7}	5.87×10^{8}	5.5581	13.2453
4-LSTM-500	2h 54m	71m 20s	1.09×10^{3}	1.35×10^{4}	4.3551	11.7138
1-CNN-150	0h 19m	08m 16s	5.3959	5.3458	3.4171	17.1287
4-CNN-150	0h 33m	15m 53s	5.7568	5.4779	3.5433	16.2600
1-MLP-500	1h 38m	08m 39s	2.2506	1.5636	1.8200	4.2648
1-MLP-2500	1h 51m	07m 01s	2.1298	1.6228	1.6589	4.4684
4-MLP-500	3h 19m	10m 05s	9.8390	66.6996	4.1207	6.6891
GBT-150	0h 30m	5m 01s	3.7928	2.1445	3.3095	6.3616
GBT-250	0h 38m	4m 34s	3.6181	2.0607	3.1275	6.1395

The right four columns of Table 3.6 summarize the inference errors defined in Equation (3.10). This metric is used, instead of the other ones presented in Section 3.4.4.2.1 as it was found to produce the best results for the Rule-Based classifier in stage 2.

Please note IPC inference is a regression task, not classification. Thus, classification metrics, such as true positive and false positive rates, do not apply here. The results are averaged across all probes for bug-free designs of microarchitectures in Set IV. Median errors of different ML engines are close to each other. LSTM-based implementations have huge variance and average errors due to a few non-convergent outliers. Figure 3.7 shows the average values of the IPC inference error (according to Equation (3.10)) for every probe in multiple machine learning engines across the architectures used for testing. Error values were sorted in order to ease the visualization of the picture. The y-axis of the picture was cropped at 25 in order to show the results more clearly.

However, several LSTM data points are far greater than that value and lie outside of the plot.

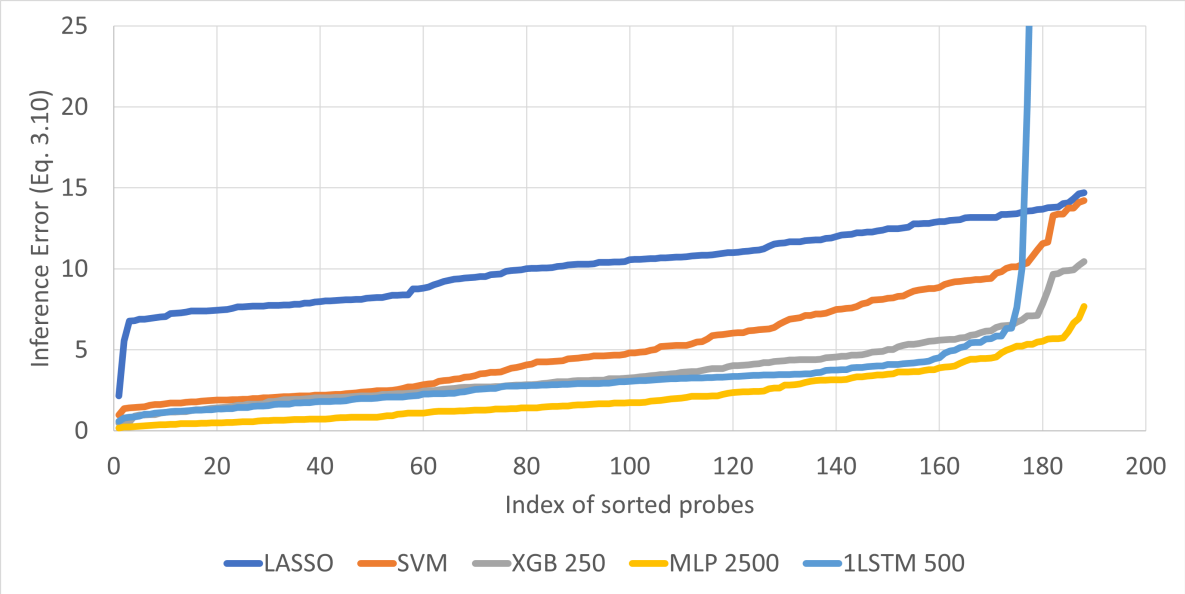
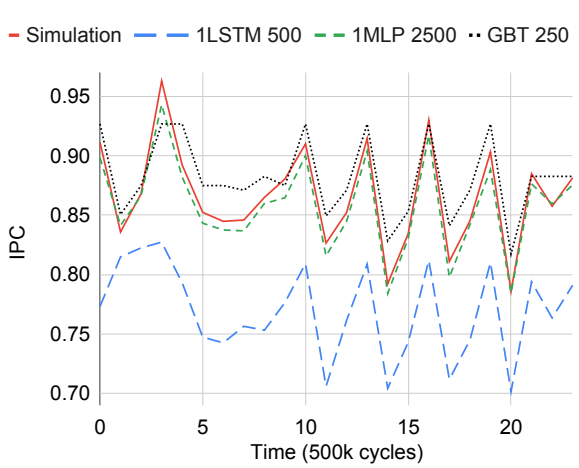


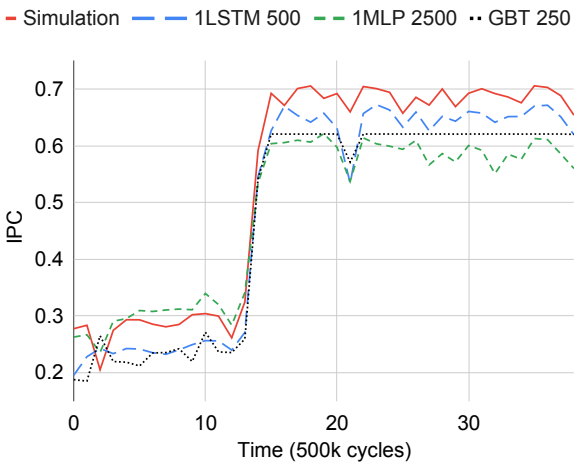
Figure 3.7: Distribution of IPC inference error for different ML engines.

Although input features have a time series format, where LSTM is expected to excel, the actual LSTM errors here are no better than non-recurrent networks and sometimes much worse. There are two reasons for this. First, the chosen time step size is large enough such that the history effect has already been well counted in one time step and thus the recurrence in LSTM becomes unnecessary. Second, LSTM is well-known to be difficult to train and the difficulty is confirmed by those outlier cases. In stage 2 of the methodology, LSTM results with huge errors are not used.

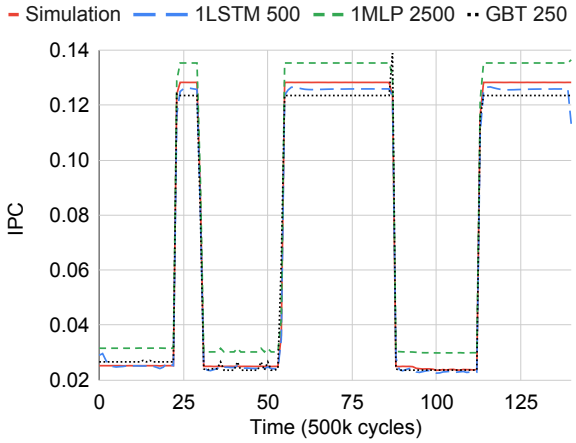
Figure 3.8 displays the IPC time series for three SimPoints (chosen to represent varied behavior), on the Skylake microarchitecture (which is part of the test set), here the red solid lines indicate the measured (simulated) IPCs, while the dotted/dashed lines are IPC inferences of the different ML engines. In general, the ML models trace IPCs very well. LSTM has relatively large errors in Figure 3.8a, yet it still shows strong correlation with the simulated IPCs. In all three cases the results confirm the effectiveness of the machine learning models across various scenarios.



(a) SimPoint #12 of gcc



(b) SimPoint #16 of bzip2

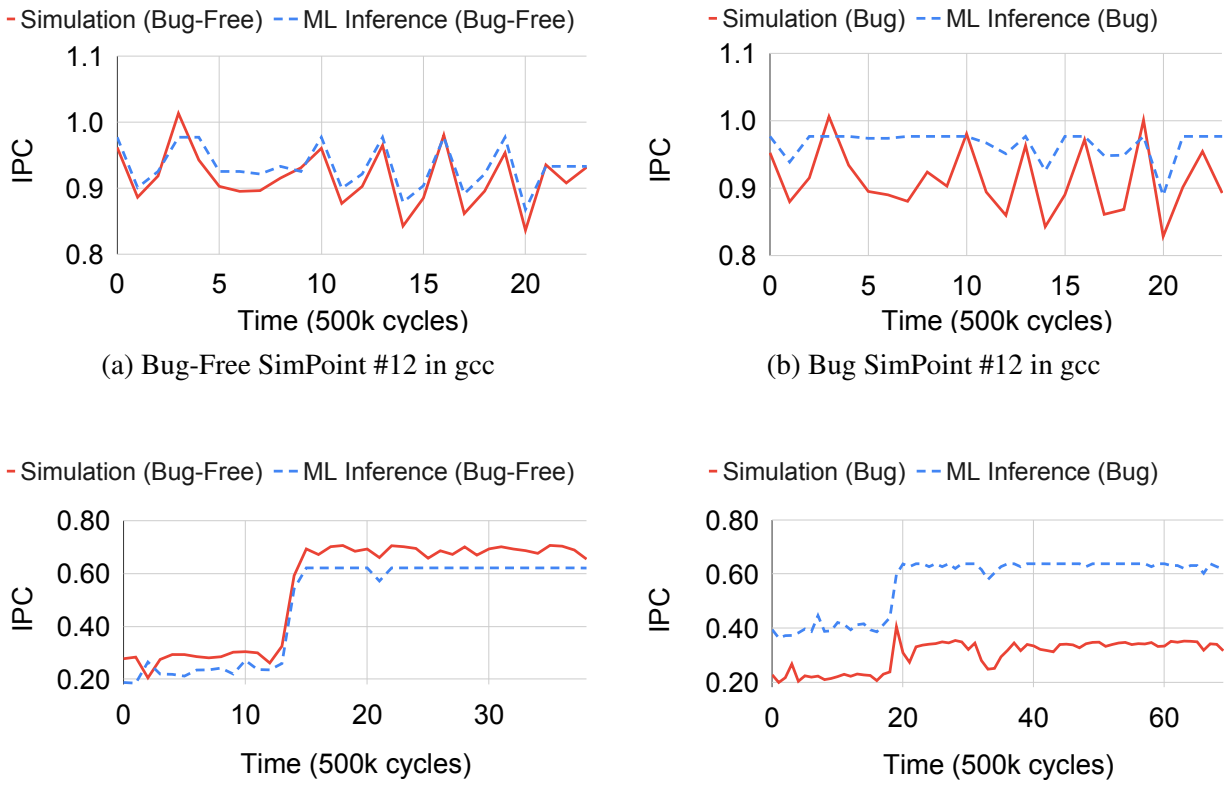


(c) SimPoint #1 of cactusADM

Figure 3.8: Examples of ML-based IPC inference and simulated IPC on bug-free microarchitectures.

Although IPC inference accuracy for bug-free designs is important, the inference error difference between cases with and without bugs matters even more. Such difference is illustrated by two SimPoints in Figure 3.9. In Figure 3.9a, where the microarchitecture is bug-free, GBT-250 estimates IPCs very accurately. When there is a bug, however, as shown in Figure 3.9b, the inference errors drastically increase. The same discrepancy is also exhibited between another bug-free design, Figure 3.9c, and buggy design, Figure 3.9d. Both the examples demonstrate that a significant

loss of accuracy implies bug existence.



(a) Bug-Free SimPoint #12 in gcc

(b) Bug SimPoint #12 in gcc

(c) Bug-Free SimPoint #16 in bzip2

(d) Bug SimPoint #16 in bzip2

Figure 3.9: Comparison of IPC estimations between microarchitectures with and without performance bugs obtained by using ML model GBT-250.

In particular, the differences between Figures 3.9c and 3.9d are worth further discussing. As it can be seen in Figure 3.9c the represented SimPoint is composed by two different "phases", differentiated by their IPC. In Figure 3.9d those two phases behave in different manners. In the first phase, the IPC on the buggy case is only slightly smaller than in the bug-free case, however, the ML model is predicting a much higher IPC. This captures the case where, although the impact on IPC is not significant, given the changes on the performance counters due to the bug, the model expects that the performance would be much higher than what is actually obtained. In the second phase, the IPC in the buggy case is significantly degraded, however, the ML model predicts how

much the IPC should be on a bug-free case and produces inferences that are almost identical to the IPC for the bug-free case on that second phase. All this is achieved because the ML models were able to learn the relationships between IPC and performance the selected performance counters.

3.4.6.2 Bug detection accuracy

In this section, the evaluation results for stage 2 classification are shown. The evaluation includes results for both rule and ML-based mechanisms, as well as comparisons with the naïve single-stage baseline approach described in Section 3.4.3. For the single stage approach the results use Gradient Boosted Trees engine with 250 trees, as it provides the best results. Its model features include simulated IPCs in addition to data from the selected counters and microarchitectural design parameters. However, it uses a single value for each feature aggregated from an entire SimPoint instead of the time series.

The bug detection efficacy is evaluated by the metrics below.

$$\text{FPR} = \frac{\text{FP}}{\text{N}}, \quad \text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{TPR} = \frac{\text{TP}}{\text{P}} \tag{3.15}$$

where N and P are the number of real negative (no bug) and positive (bug) cases, respectively. FP (False Positive) indicates the number of cases that are incorrectly classified as having bugs. TP (True Positive) is the number of cases that are correctly classified as having bugs. Additionally, ROC (Receiver Operating Characteristic) AUC (Area Under Curve) is evaluated. ROC shows the trade-off between TPR and FPR. ROC AUC value varies between 0 and 1. A random guess would result in 0.5 and a perfect classifier can achieve 1. Accuracy, another common metric, is not employed here as the number of negative cases is too small, i.e., the testcases are imbalanced.

The detection metrics of performance bug detection for the single-stage baseline, as well as both stage 2 methodologies are shown in Table 3.7. Different ML engines that were used in stage 1 are listed in the "Stage 1 ML Model" column. The data in Table 3.7 assumes that the available data to train the models comes from bug-free legacy architectures.

From the table it can be seen that the Rule-Based methodology with GBT-250 ML model

Table 3.7: Bug detection results for multiple ML engines when exclusively bug-free train data is used.

Stage 2 Method	Stage 1 ML Model	FPR	TPR	ROC AUC	Precision	TPR for different bug categories			
						High	Medium	Low	Very Low
Single-stage baseline	N/A	0.00	0.75	0.87	1.00	1.00	0.71	0.74	0.41
Rule-Based	**Lasso**	0.20	0.39	0.57	0.58	0.74	0.14	0.17	0.23
Rule-Based	**1-LSTM-150**	0.43	0.68	0.83	0.78	0.98	0.71	0.37	0.56
Rule-Based	**1-LSTM-250**	0.34	0.67	0.80	0.81	0.98	0.79	0.54	0.33
Rule-Based	**1-LSTM-500**	0.25	0.68	0.80	0.86	1.00	0.93	0.41	0.51
Rule-Based	**4-LSTM-150**	0.00	0.45	0.73	1.00	0.89	0.71	0.15	0.05
Rule-Based	**4-LSTM-500**	0.25	0.56	0.80	0.75	0.94	0.57	0.27	0.33
Rule-Based	**1-CNN-150**	0.75	0.68	0.71	0.66	0.89	0.71	0.46	0.62
Rule-Based	**4-CNN-150**	0.34	0.53	0.63	0.66	0.87	0.57	0.27	0.31
Rule-Based	**1-MLP-500**	0.50	0.84	0.88	0.81	1.00	1.00	0.83	0.59
Rule-Based	**1-MLP-2500**	0.48	0.80	0.85	0.80	0.98	0.93	0.73	0.59
Rule-Based	**4-MLP-500**	0.43	0.77	0.77	0.81	0.96	0.86	0.63	0.62
Rule-Based	**GBT-150**	0.00	0.80	0.89	1.00	0.96	0.93	0.73	0.59
Rule-Based	**GBT-250**	0.00	0.84	0.90	1.00	1.00	0.93	0.69	0.75
ML-Based (SVM)	**GBT-250**	0.00	0.34	0.75	1.00	0.76	0.36	0.12	0
ML-Based (RF-5)	**GBT-250**	0.02	0.36	0.69	0.98	0.76	0.21	0.12	0.11

is the best performing, beating both single-stage baseline, as well the ML-based methodology. This method is capable of detecting 91.5% of the bugs with IPC impact greater than 1% without incurring in false-positives.

The ML-based methods (both SVM, as well as RF) perform very poorly on the evaluated bugs. The results obtained for GBT are not included on the table, as they are even worse to those of RF. In particular, it was also found that when more trees are used on this method, the results suffer heavily, due to over-fitting.

Previous architectures that are considered "bug-free" may actually have performance bugs, so the case when designs with a bug are presumed as "bug-free" and are used for training is studied. The results are shown in Table 3.8. The two bugs used for the table correspond to bugs with low IPC impact across the studied workloads. These are chosen as representative cases, since bugs with higher IPC impact will likely be caught during post-silicon validation of previous designs and will be fixed by the time a new microarchitecture is being developed. To evaluate these, the GBT-250 model for stage 1 of the methodology is used, as it was best performing. As expected, these results

show some degradation in detection when buggy data is used. However, it is still capable to detect around 70% of the bugs, while incurring in a few false-positives. The ML-based methodology is not evaluated here, as their results are very poor, even when only bug-free architectures are used to train the models.

Table 3.8: Bug detection results for multiple ML engines when data from buggy designs is used to train. Bug 1 refers to "If XOR is oldest in IQ, issue only XOR" and Bug 2 is "If ADD uses register 0, delay 10 cycles".

Bugs in presumed bug-free training	Stage 2 Method	Stage 1 ML Model	FPR	TPR	ROC AUC	Precision	TPR for different bug categories			
							High	Medium	Low	Very Low
No Bug	Rule-Based	**GBT-250**	0.00	0.84	0.90	1.00	1.00	0.93	0.69	0.75
Bug 1	Rule-Based	**GBT-250**	0.15	0.69	0.83	0.90	0.94	0.31	0.61	0.54
Bug 2	Rule-Based	**GBT-250**	0.20	0.74	0.84	0.88	0.98	0.43	0.63	0.60

It is important to note that the tradeoff between true and false positive detection varies depending on the IPC impact caused by each bug. Figure 3.10 shows several examples of how the ROC curve looks for different bug types when stage 1 uses GBT-250 model and a rule-based classifier for stage 2 is used. Difficult to catch bugs usually have a lower ROC AUC, while other bugs with higher IPC impact can be detected without false-positives.

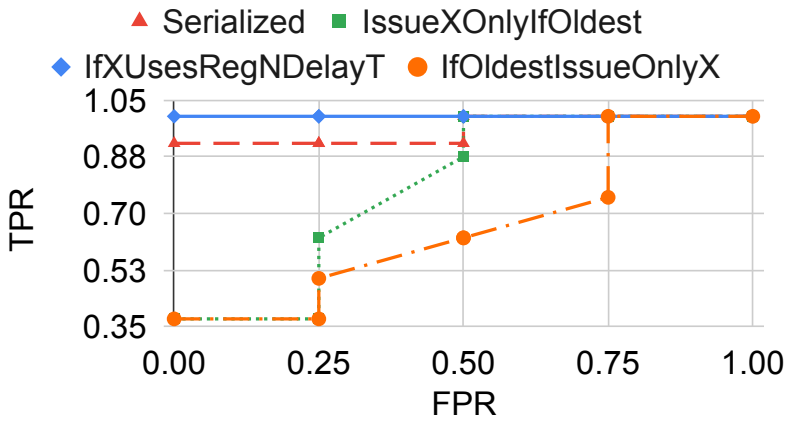


Figure 3.10: ROC curves for GBT-250 on different bug types using rule-based classifier.

Since it was found that the rule-based methodology for stage-2 using GBT-250 in stage 1 was best performing, in the upcoming sections, that configuration is used, unless otherwise is specified.

3.4.6.3 Impact of the number of probes on bug detection

A trade-off associated with the number of probes exists for the accuracy of the proposed methodology. More probes can potentially detect more bugs or reduce false positives, at the cost of higher runtime. To see how accuracy varies with probe count an examination of the impact of reducing the number of probes is conducted.

This experiment is performed using an iterative approach. Every iteration five probes are removed, and the models are re-trained with the reduced set and its detection metrics are collected. Results are shown in Figure 3.11. The probe reduction is analyzed two with different discarding orders.

1. Remove the probes with the highest error in IPC inference first. The insight for using this method is, if a probe has high IPC inference error, it is likely that the model is not very accurate, and therefore, the information it provides is not reliable.

2. Remove probes in random order. This case is equivalent to having fewer probes with which to test the design. This method was executed 100 times and the results correspond to the average value obtained, this was done to mitigate the effects of the randomness of the procedure.

Results for both orderings, show that, when the number of probes is reduced, the quality of results is degraded, either by an increase in FPR or by decreasing the TPR. However, the accuracy degrades very slowly. This indicates that the methodology is quite robust, and even with fewer benchmarks the methodology could be used with little impact on the quality of the results.

Reducing the number of probes that need to be executed would help to speed the bug detection process, not only on the probe execution, or simulation time, but also on the machine-learning training and inference runtime. However, it is important to avoid removing probes that provide

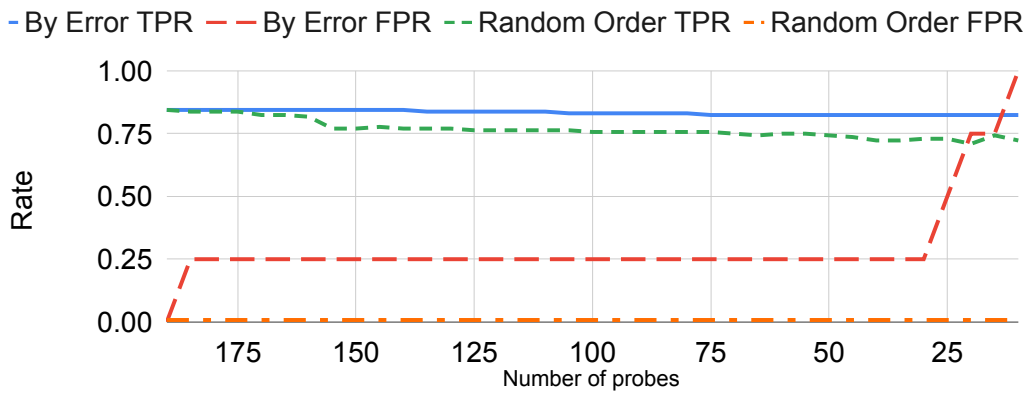


Figure 3.11: Impact of the number of available probes on the bug detection results.

coverage that is not given by any other probe on the set, since this will decrease the number of bugs that the method would be able to detect.

3.4.6.4 Impact of the counter selection mechanism on bug detection

The counter selection methodology discussed in Section 3.3.1.2 is evaluated by comparing its results versus a set of 22 manually, but not arbitrarily, selected counters, these include miss rates for different cache levels, branch statistics and other counters related to the core pipeline and how many instructions each pipeline stage has processed. All these correspond to performance counters that provide relevant data for performance estimation.

Unlike the proposed counter selection method, here the same 22 counters are used for all the probes. The results are evaluated for models 1-LSTM-500 and GBT-250 in stage 1 and a rule-based classifier in stage 2. The obtained results can be seen on Figure 3.12.

The proposed methodology achieves better results in both machine learning models when compared to the results obtained by a manual counter selection. Despite being heuristic, the automated counter selection methodology generally facilitates better detection results.

In average the IPC inference errors obtained by the models using the automatically generated counter list are $\sim 3\times$ lower than the errors obtained by using the hand-picked methods. No considerable difference in training or inference runtime exists between the methods.

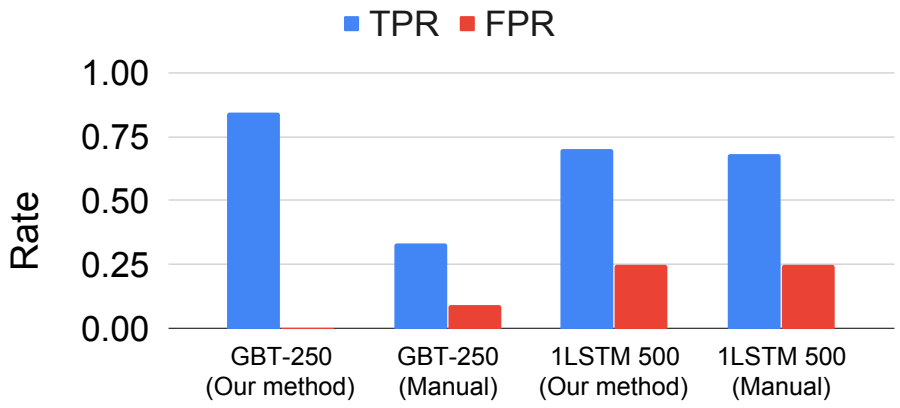


Figure 3.12: Impact of the counter selection method on the bug detection results.

3.4.6.5 Impact of the time step size on bug detection

In the IPC modelling stage, the input features are taken as time series with each time step being 500K clock cycles long. Experiments were performed to observe the effect of different time step sizes. The results of the IPC inference errors obtained for different time steps are plotted in Figure 3.13.

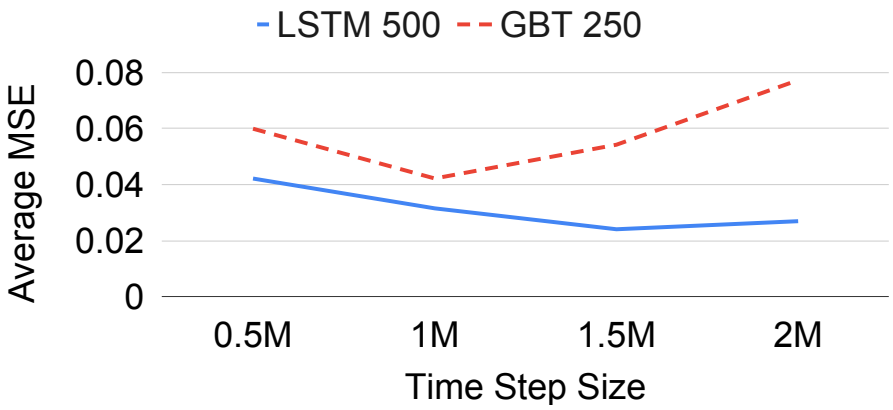


Figure 3.13: Impact of the time step size on bug detection measured by the average MSE across all probes.

It can be observed that, when the time step size increases, inference errors for model 1LSTM-

500 decreases. This is because a coarser grained inference is generally easier than a fine-grained. MSE is used in the picture, as opposed to the error defined by Eq. (3.10) as this metric is not comparable among different step sizes.

The reduced IPC inference errors do not necessarily lead to improved bug detection results. In fact, Figure 3.14 shows that both TPR and FPR degrade as the time step size increases. The rationale is that whether or not the IPC inference is sensitive to bugs matters more than the accuracy. The results in both figures confirm the choice of 500K cycles as the time step size.

Besides the efficacy of bug detection, time step size also considerably affects computing runtime and data storage. For this work, it was determined that a step size of 500K clock cycles reaches a good compromise between bug detection and computing load.

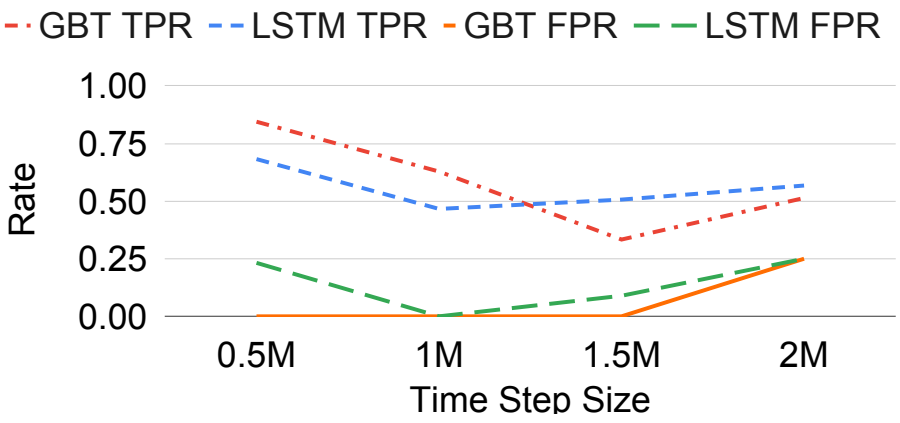


Figure 3.14: Impact of the time step size on bug detection measured by the TPR and FPR.

3.4.6.6 Impact of the window size on bug detection

The IPC inference in stage 1 can take the feature data from a series of time steps. So far in this work, the window size used for all the experiments is zero (only the current time step information is provided to the model). This is because the decided time step size is sufficiently large. Experiments were conducted to observe the impact of increasing the window size. Table 3.9 shows the TPR and FPR obtained when the window size is increased.

Table 3.9: Impact of the window size on performance bug detection.

	Window Size			
	0	1	2	3
TPR	0.84	0.48	0.32	0.48
FPR	0.00	0.21	0.00	0.39

The results confirm the choice of a window size of zero throughout the methodology evaluation. Given the time step size, the results suggest that adding information from previous time steps do not help to increase sensibility to performance bugs, furthermore, it actually degrades it.

3.4.6.7 Impact of the usage of microarchitecture design parameter as features on bug detection

In the proposed methodology we make use of microarchitectural design parameters/specifications (*e.g.* ROB size, issue width, etc.) as static features for the IPC modeling stage. Here, the impact of removing these static features on the accuracy of the bug detection methodology is evaluated. Figure 3.15 shows the obtained results.

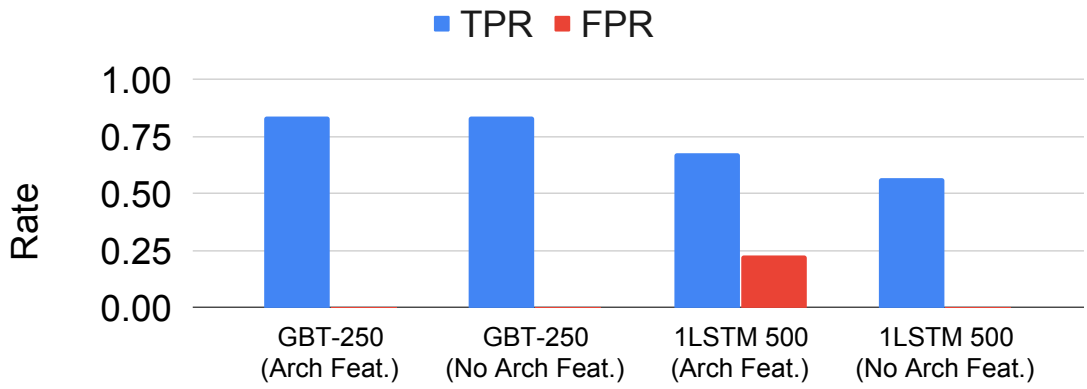


Figure 3.15: Impact of the usage of microarchitecture design parameter as features on bug detection.

The results show that removing the design parameters has no impact on the accuracy of the GBT-250 models and produces only a small reduction on the number of detected bugs with the

1LSTM-500 model, although the number of false alarms is also reduced.

By further evaluation, it was found that this impact is contained within the bugs of *Low* or *Very Low* IPC impact. These results indicate that performance impact information is, in many cases, sufficiently contained within the performance counters (*i.e.* performance counter data inherently conveys enough information for the model to infer the IPC of different microarchitectures on the given workloads), and the change on microarchitectural specifications has a very small impact on the quality of results for this methodology.

3.4.6.8 Impact of the number of training microarchitectures on bug detection

The effect of the number of available architectures to train the models of the methodology is also evaluated. Here, five microarchitectures are used to train the IPC models (Set I), instead of nine. Sets II and III were reduced from three and four to two and three microarchitectures, respectively. In each case the "artificial" microarchitectures were dropped, keeping only the real ones. The number of testing microarchitectures is kept constant, the results are shown in Figure 3.16.

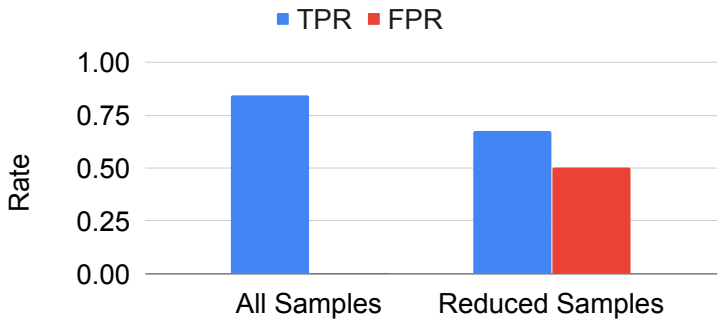


Figure 3.16: Impact of the number of training microarchitectures on bug detection.

From the obtained results is confirmed that creating artificial architectures is necessary in order to augment the data sets. This aids the models to learn the difference between performance variation due to microarchitectural specifications and performance bugs.

3.5 Machine learning for performance bug localization

Detecting whether a design has a bug or not is only the first step in the debugging process. In order to provide meaningful feedback to the designers, it is also important to determine in which part of the microprocessor design the performance bug is occurring. This problem is significantly more challenging than bug detection, as it requires to localize the microarchitectural unit causing the performance bug.

3.5.1 Problem formulation for the bug localization task

The *goal* of this work is to provide the microarchitecture designer with information regarding the units that are likely to have performance bugs. The proposed methodologies assume the availability of legacy architectures, both bug-free designs, as well as designs where a performance bug has been found (either by manual debug or using automated methods like the presented in Section 3.4) and identified are necessary. This assumption generally holds, given the thorough pre- and post-silicon debug to which the legacy designs are submitted. In every design with performance bugs, it is assumed that only one bug is present at a time. It is also assumed that there is not a large architectural shift between the legacy designs and the designs being debugged. When such thing occurs, the methodology can still be partially utilized for bug localization on common areas between designs or by using workloads that do not exercise new functionalities. Although in this work, the techniques are evaluated using pre-silicon simulations, the techniques proposed here can be used in post-silicon debugging as well.

The bug localization problem is formulated as a multi-class classification problem. Every microarchitectural unit where the bug might be located corresponds to a class $u_j \in U$, where U is the set of all units. To solve the multi-class classification task, some common strategies in the ML community are:

- *One-vs-one*: In this strategy a base classifier is trained to distinguish between all the possible pairings of classes. So, a total of $|U| \cdot (|U| - 1)/2$. When a prediction is performed, a voting mechanism is used and the class with the highest number of votes is used as output of the

overall multi-class classifier [84].

- *One-vs-all*: Also known as one-vs-rest. In this strategy a single classifier per class is trained (for a total of $|U|$ base models). Each one learns to predict if a sample belongs to a particular class or not. Here, classifiers that produce confidence scores, as opposed to straight binary outputs, are needed. When a prediction is performed, the model with higher confidence is used as output of the overall multi-class classifier [85].

- *Multi-output classifier*: In this strategy, a single model with multiple outputs is trained, each output represents a possible class [86]. This can be easily achieved on tree-based methods or neural networks, where the final layer has multiple outputs followed by a softmax function to normalize the values to probabilities.

3.5.2 Methodology overview for the bug localization task

In this work, two machine learning-based methodologies are proposed. The first methodology uses multiple machine learning models, each of which classifies if a unit contains the bug. Then, results of these models are aggregated to obtain an ordered list of units according to their confidence levels for bug existence. This methodology is referred to as **"Counter-Based Classification"** or **"CBC"** and it is discussed in detailed in Section 3.5.3. The second methodology is a two-stage approach. Its first stage includes machine learning models for predicting performance in terms of IPC and the models are trained with bug-free designs (as described in Section 3.4.4.1). In the second stage, prediction errors of these models are utilized for estimating the likelihood of bug existence for each unit. The name **"Performance Prediction error-Based Classification"** or **"P2BC"** is used to refer to this method and it is discussed in detail in Section 3.5.4. The output from either of the methodologies provides a priority list of microarchitecture units for further analysis by designers.

Both methods rely on the usage of performance probes, the procedure to design these is the same as the one presented in Section 3.3.1. Ultimately the same set of probes as the ones used for performance bug detection (discussed in detail in Section 3.4.5.1) is used for the performance bug

localization task.

3.5.3 Performance bug localization via Counter-Based Classification (CBC)

The methodology proposed in this section consists of a single machine learning stage followed by an aggregation procedure. An overview of this methodology is shown in Figure 3.17.

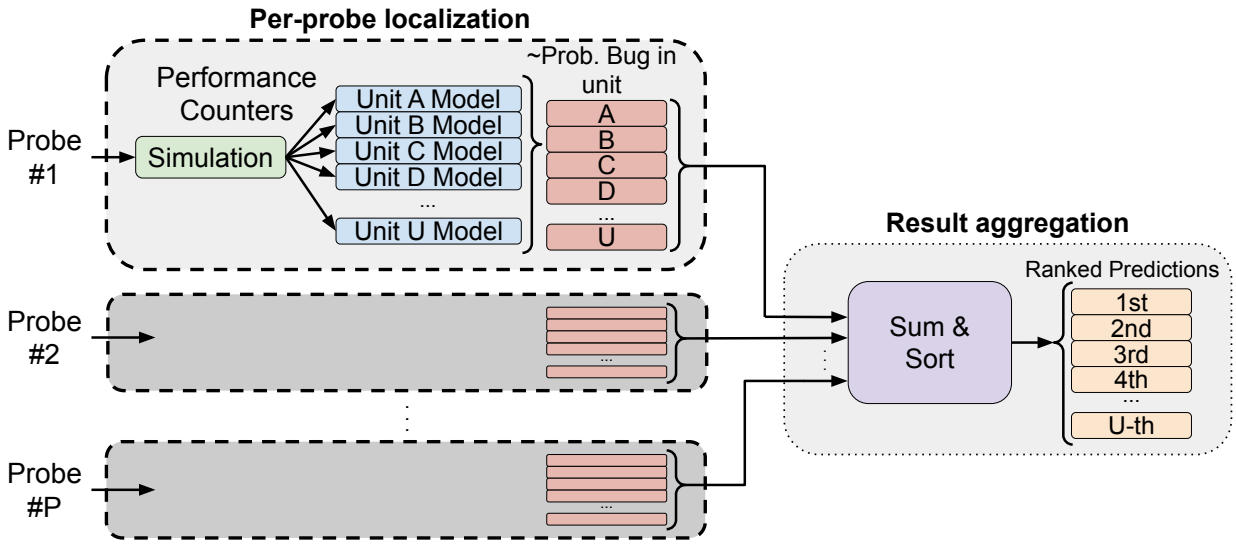


Figure 3.17: Overview of the CBC methodology for bug localization.

As far as the multi-class strategy the usage of a multi-output model was evaluated, however, it did not achieve the necessary accuracy. Therefore, a one-vs-all strategy is followed as it provides the best compromise between accuracy and number of base classifiers needed. Here, a base classifier is a model to classify if the bug exists in a specific unit for a probe. If the set of all probes is denoted by P, the base classifier for probe $p_i \in P$ that flags the bug in unit u_j is denoted as $m_{i,j}$. Following this, a total of $|P| \cdot |U|$ base classifiers are needed.

Although different performance counters were selected for each probe (as discussed in Section 3.4.5.1), this methodology performs better when the models use the super-set composed by the union of the counters selected for all the probes. This is because, for some probes, the selected counters do not contain any information regarding specific units that might be affected by the per-

formance bug. Providing every probe with a larger counter set increases the visibility the models have in all the units of the system. Although using the union super-set significantly increases the number of features per model by a factor of $\sim 15\times$, this merged set is still about $10\times$ smaller than the complete list of available counters.

Each of the base classifiers $m_{i,j}$ produces a real-valued confidence score, which is a soft classification in $[0, 1]$ for probe p_i to tell if the bug is at unit u_j. Since the confidence scores from all classifiers of a probe do not add up to 1, they do not represent probability. However, they serve as probability proxies, as high scores mean high probability of the bug being in that unit. The confidence scores for every unit across every probe are summed to create a final score for each unit. Finally, these scores are sorted, and larger values indicate a higher chance that the bug is present in that unit. The overall CBC output is a list of most likely units where the bug might be located.

In order to train base model $m_{i,j}$, one or more legacy architectures with and without performance bugs are used. Samples from architectures with bugs occurring at unit u_j are considered "positive" cases for the model, while any other unit is considered a "negative" case.

The input data from the performance counters is used in a time-series format, with their values being sampled and reset every time a determined number of cycles have passed (*e.g.* every 100k cycles). Resetting the counters ensures that their value reflects only the behavior of the counter in the current time-step, without keeping track of its history. Because of this data format, two different methods to calculate the confidence score are evaluated and contrasted:

1. *Per-trace classification*: Using neural networks that are able to take advantage of temporal locality, such as Convolutional Neural Networks [80] or Long Short-Term Neural Networks [81], the complete time trace can be processed and a single score value generated in the end. The proposed methodology uses CNNs, as LSTMs are very hard to train due to its recurrence and did not produce satisfactory results.

2. *Per-time-step classification*: At every time-step t_i, a bug location prediction is performed by using the input features from a window of W time-steps $t_i, t_{i-1}, ..., t_{i-W+1}$. With this, a classification result per time-step can be obtained, ultimately creating a "classification

trace". A window of one time-step represents the case where only data from the current time-step is used to predict the bug location, while larger windows consider the counter history. This method allows for other machine learning methods to be used, such as Multi-Layer Perceptrons [27], Random Forest [24] or Gradient Boosted Trees [87]. Although all these methods were evaluated, the proposed methodology uses Gradient Boosted Trees, as it was found the best performing. Ultimately, to transform the prediction trace into a single value, the mean value across the whole time trace is calculated.

3.5.4 Performance bug localization via Performance Prediction error-Based Classification (P2BC)

This method is composed by two different machine learning stages. An overview of this is shown in Figure 3.18.

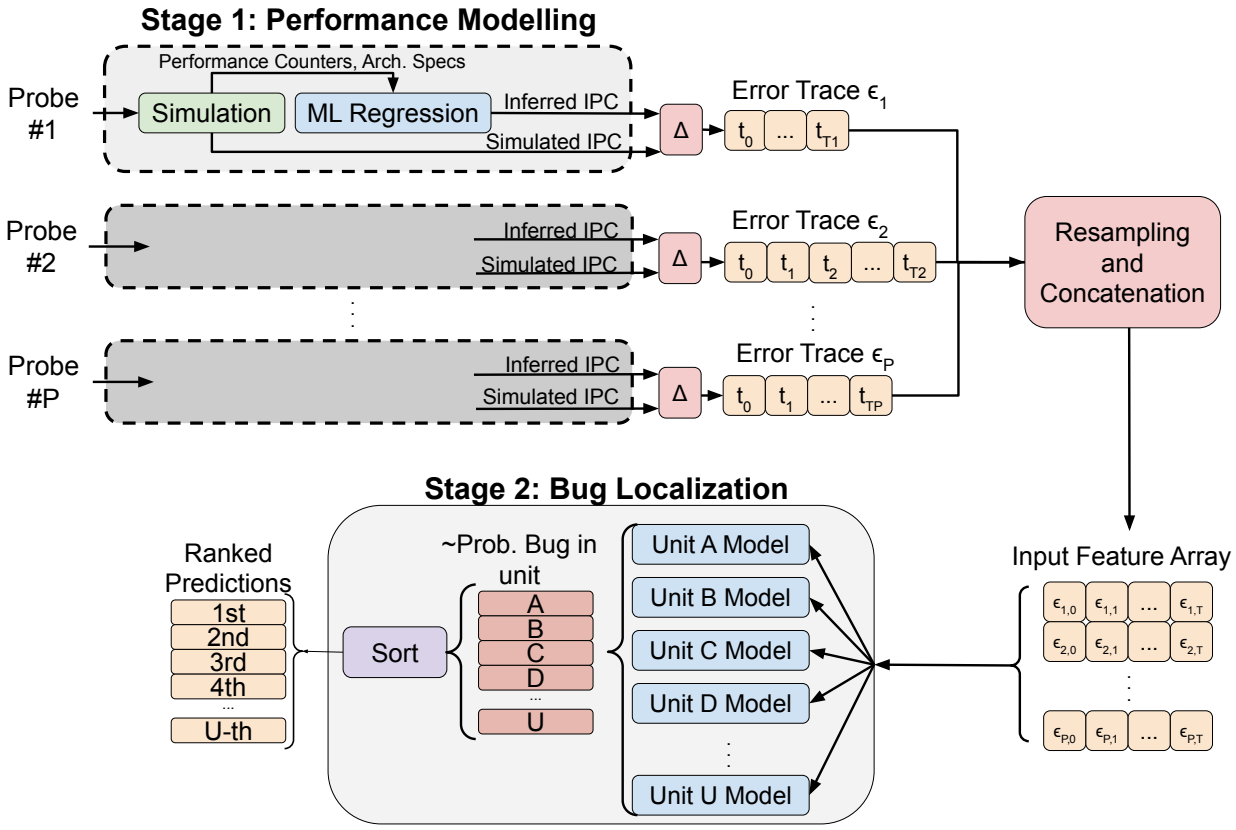


Figure 3.18: Overview of the P2BC methodology for bug localization.

Its stage one is a set of performance (IPC) prediction models trained with only bug-free design data so as to capture the relationship between counter data and performance under healthy conditions. When such models are applied on buggy designs, significant prediction errors show up as the healthy conditions no longer hold. This stage is based on the work developed for automatic performance bug detection described in Section 3.4.4.1. As such, the ML models for performance prediction use gradient boosted trees [87] as it was demonstrated to be the best performing engine for this task. Please note the performance models here do not mean to be golden references and in fact they perform poorly for buggy designs. However, the prediction errors contain useful information and serve as features for the second stage classifiers. Hence, P2BC localizes bugs according to symptoms of performance model failures.

3.5.4.1 Stage 1: Performance modeling

Stage one of this methodology aims to model bug-free processor performance by inferring a target metric (IPC in this case) using performance counters as input features to a set of machine learning models. There is one ML regression model for each probe and the counters of each probe are selected according to Section 3.3.1. The per-probe models help achieve higher accuracy, as they are specialized to each individual probe. As in the methodology shown in Section 3.5.3, the performance counters are used as a time-series, and therefore, an inferred IPC time trace is produced.

For this stage, the training data is exclusively from bug-free designs. Every model takes training data from a set of legacy microarchitectures. This ensures the model learns to differentiate abnormal behavior due to performance bugs vs. due to different microarchitectures.

Once the inferred IPC trace is obtained from these models, an IPC error trace ϵ_i is calculated by computing the difference between the IPC trace obtained via ML performance models and the one obtained from the simulations. After error traces are obtained from all probes, they all go through a resampling and concatenation procedure.

The goal of this procedure is to arrange the error traces in a uniform bi-dimensional array so that ML strategies like convolutional neural networks can be used. Different traces have different

lengths, as such, the resampling procedure is needed to achieve uniformity. This procedure uses the Fourier methodology implemented on the SciPy library [88].

This resampling procedure is based on Nyquist-Shannon sampling theorem [89] which states that the sampling frequency required to capture all the information from a signal is two times the maximum frequency present on such signal. The idea is that, by modifying the frequency domain spectrum of a signal, the number of samples needed to capture all the trace information can be changed as well. Depending on the number of samples in the trace, there can be two possible resampling mechanisms:

- **Down-sampling**: This is when the number of available samples is greater than the target. In this case, after an FFT is applied, the frequency spectrum is truncated at the required maximum frequency. With this, the signal can be rebuilt with less samples by using FFT^{-1}.
- **Up-sampling**: This is when the number of available samples is less than the target. Here, after an FFT is applied, the frequency spectrum is zero-padded. Then, the signal can be rebuilt with more samples by using FFT^{-1}.

This process is illustrated by Figure 3.19. Empirical results demonstrated that the best results can be achieved when the target number of samples is equal to the average number of time-steps across all the probes. The resulting array is used as input for stage two, described below.

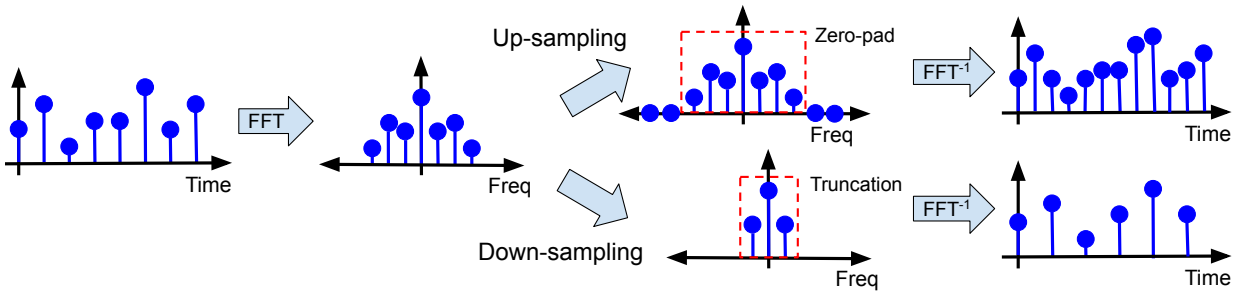


Figure 3.19: Fourier-based resampling methodology.

3.5.4.2 Stage 2: Error-based bug localization

In this stage, a multi-class classifier is trained to identify the microarchitectural unit where a performance bug might be located. A "one-vs-all" methodology is followed, same as in Section 3.5.3.

To train these models, IPC inference errors obtained by stage one in legacy architectures are used. Samples from bug-free designs, as well as designs with bugs in different locations are used. In a similar fashion to the models in CBC, when the model to classify bugs in the class $u_j \in U$ is trained, the samples from architectures with bugs occurring at unit u_j are considered "positive" samples, while any other unit is considered "negative".

When an inference is performed, the confidence score of every model is sorted, and the units from the more confident models are considered the most likely locations of the bug.

The classifiers are implemented using one-dimensional convolutional neural networks. The convolution operations are performed exclusively along different time-steps of the same probe, and every probe is used as a different channel [80]. The reason to do this is because there is no relevant information to be learned across same time-steps of different probes given that the probe order is completely arbitrary.

3.5.5 Trade-offs between CBC and P2BC

Each of these methodologies have their advantages and drawbacks, these are discussed as follows:

1. **Data storage and runtime**: The CBC approach requires a significantly higher number of models than the P2BC method. For the former, one model per unit per probe is needed, $|P| \cdot |U|$ in total, while the latter requires one model per probe for IPC estimation and one model per unit for bug localization, for a total of $|P| + |U|$. The increased number of models in CBC creates a larger data storage requirement ($\sim 100\times$ in this evaluation), as well as a longer runtime for training and inference (between $6\times$ and $10\times$). However, in both methodologies, the training of all the models can be achieved within a day, while the

inference time is in the order of a few minutes, without accounting for speedup that might be gained by parallelization.

2. **Accuracy**: It was found that CBC performs better than P2BC for bugs with average IPC impact greater than 1%, but the roles are swapped for smaller impact bugs (details in Section 3.5.8.3). One reason for this is that in P2BC, the simulation results are compared against a "bug-free baseline" in stage one. This is likely making the models more sensitive to small performance perturbations. On the other hand, for larger bugs, having more models allows for the direct classification approach to learn to discriminate the different behaviors.

3. **Incremental probe addition**: CBC is more friendly to incrementally adding a new probe. To add a new probe, $|U|$ new models must be trained for this method. Although for P2BC the number of new models is not significantly larger, only $|U| + 1$ (one IPC model, and $|U|$ models in stage two), adding a new probe involves re-training of the entire stage two, which could significantly impact the quality of results and requires a longer training period.

4. **Incremental bug unit location addition**: Both approaches allow for incremental addition of microarchitectural units where bugs might be located. This is useful for cases when a bug location that was not part of the list of classes considered by the methodologies needs to be included, or when a new structure is added to the system. In this case, only one new model would need to be trained for P2BC, while $|P|$ would be needed for CBC. Although re-training the models for other units using samples with bugs in the new one is not required, doing so might improve the accuracy of the overall debugging approach.

3.5.6 Ensemble of methods

Although both methodologies provide satisfactory results, they don't excel in the same cases. In order to take advantage of the strengths of both methods, a simple ensemble scheme is presented in this section. The procedure is as follows.

1. The final confidence scores of each methodology are normalized, so that for every sample,

the sum of the confidences across all the units u_j equals one. Since a "one-vs-all" methodology is used, this cannot be guaranteed beforehand without this step.

2. For each unit, the average score obtained across both methodologies represent its final score.

3. This newly calculated scores are ranked, and their order is used to determine the unit with the highest probability to have a performance bug.

3.5.7 Experimental setup for bug localization

This section elaborates on the implementation and evaluation setup of the proposed methodologies. Sections 3.5.7.1 covers the details of the performance probes utilized to test the methodologies. Section 3.5.7.2 provides details on the implementation of different microarchitectures used to evaluate the techniques and Section 3.5.7.3 details the bugs injected into the simulations in order to evaluate the bug localization techniques proposed here.

3.5.7.1 Probe setup

The performance probes used for the evaluation of the bug localization mechanisms correspond to the same 190 SimPoints extracted from SPEC CPU2006 applications used for bug detection and described in Section 3.4.5.1. However, in this case, the performance counters are sampled with a finer granularity, every 100k cycles (as opposed to the 500k cycles used for bug detection). Finer granularity allows for more detailed information on the behavior of the probe, however, it comes at the cost of an increased data storage, as the size of the files generated by the simulator are significantly larger.

3.5.7.2 Simulated architectures

As in the case of performance bug detection, all the experiments performed to evaluate the proposed bug localization methodologies are based on simulations in gem5 [33] using the out-of-order core model (O3CPU) and x86 ISA in system emulation mode. Although all the evaluations are based on simulations, the methodologies can also be applied in post-silicon debugging by

using longer application runs that would only be feasible to achieve in silicon and by relaxing the sampling frequency on the performance counters to rates feasible to achieve in silicon.

The gem5 simulator is configured to emulate a diversified set of different microarchitectures to train and test our scheme. The core related settings modified to achieve this are: clock period, pipeline width, branch predictor, and the sizes of re-order buffer, load/store queue, and instruction queue. Cache related configurations (size, associativity, latency, and number of levels), as well as functional unit characteristics (count, latency, and port organization) are also modified to map the simulator behavior to the emulated core. Other microarchitectural differences besides these are not considered.

In total, 23 different microarchitecture configurations were implemented, 14 of them are based on multiple Stock-Keeping Unit (SKU) variants of the Intel Core microarchitectures: Sandy Bridge, Ivy Bridge, Haswell, Broadwell, Ice Lake and Skylake. The remaining nine were artificially created but use realistic settings. The details of the core specifications used on each case are summarized in Table 3.10, while the cache details are shown in Table 3.11. Table 3.12 described the port organization on each architecture.

The data extracted from simulations of the nine artificial microarchitectures, and the SKU variants of Sandy Bridge and Skylake are used for training purposes. The remaining four microarchitectures are used for testing the techniques. As such, the test microarchitectures are not used to train the ML models.

Table 3.10: Core specifications of the architectures used to evaluate the bug localization mechanisms.

Architecture	Clock Cycle	CPU Width	ROB Size	LSQ Size	IQ Size	Branch Predictor	Func. Unit Latency (FP / Multiplier / Divider)
Sandy Bridge	2.3GHz	4	168	42	20	Tournament	5 cycles / 3 cycles / 20 cycles
Ivy Bridge	2.0GHz	4	168	32	64	Tournament	5 cycles / 3 cycles / 20 cycles
Haswell	3.2GHz	4	192	42	20	Tournament	5 cycles / 3 cycles / 20 cycles
Broadwell	4.0GHz	4	192	42	25	LTAGE	5 cycles / 3 cycles / 20 cycles
Skylake Client DT	4.0GHz	6	224	56	64	LTAGE	4 cycles / 4 cycles / 20 cycles
Skylake Client H	3.9GHz	6	224	56	64	LTAGE	4 cycles / 4 cycles / 20 cycles
Skylake Client S	4.2GHz	6	224	56	64	LTAGE	4 cycles / 4 cycles / 20 cycles
Skylake Client U	3.1GHz	6	224	56	64	LTAGE	4 cycles / 4 cycles / 20 cycles
Skylake Client Y	2.8GHz	6	224	56	64	LTAGE	4 cycles / 4 cycles / 20 cycles
Skylake Server DE	3.0GHz	6	224	56	64	LTAGE	4 cycles / 4 cycles / 20 cycles
Skylake Server SP	3.1GHz	6	224	56	64	LTAGE	4 cycles / 4 cycles / 20 cycles
Skylake Server W	4.5GHz	6	224	56	64	LTAGE	4 cycles / 4 cycles / 20 cycles
Skylake Server X	4.3GHz	6	224	56	64	LTAGE	4 cycles / 4 cycles / 20 cycles
Ice Lake	4.6GHz	6	352	1024	70	LTAGE	3 cycles / 4 cycles / 18 cycles
Artificial 1	4.0GHz	4	256	32	64	Tournament	4 cycles / 4 cycles / 20 cycle
Artificial 2	2.5GHz	4	192	32	64	Tournament	5 cycles / 3 cycles / 20 cycles
Artificial 3	3.0GHz	8	32	32	64	LTAGE	4 cycles / 4 cycles / 20 cycle
Artificial 4	4.0GHz	2	192	32	64	Tournament	5 cycles / 3 cycles / 20 cycles
Artificial 5	3.5GHz	2	32	32	64	Tournament	4 cycles / 3 cycles / 11 cycles
Artificial 6	3.5GHz	4	192	56	64	Tournament	4 cycles / 4 cycles / 20 cycles
Artificial 7	3.0GHz	4	32	32	25	LTAGE	2 cycles / 7 cycles / 69 cycles
Artificial 8	3.0GHz	2	192	224	64	Tournament	4 cycles / 3 cycles / 11 cycles
Artificial 9	3.5GHz	4	32	32	64	Tournament	5 cycles/ 4 cycles / 30 cycles

Table 3.11: Cache specifications of the architectures used to evaluate the bug localization mechanisms.

Architecture	L1 Cache (Size / Assoc. / Latency)	L2 Cache (Size / Assoc. / Latency)	L3 Cache (Size / Assoc. / Latency)
Sandy Bridge	32kB / 8-way / 4 cycles	256kB / 8-way / 11 cycles	2MB / 16-way / 28 cycles
Ivy Bridge	32kB / 8-way / 4 cycles	256kB / 8-way / 11 cycles	2MB / 16-way / 28 cycles
Haswell	32kB / 8-way / 4 cycles	256kB / 8-way / 12 cycles	2MB / 16-way / 59 cycles
Broadwell	32kB / 8-way / 4 cycles	256kB / 8-way / 12 cycles	64MB / 16-way / 59 cycles
Skylake Client DT	32kB / 8-way / 4 cycles	256kB / 8-way / 12 cycles	8MB / 16-way / 34 cycles
Skylake Client H	32kB / 8-way / 4 cycles	256kB / 8-way / 12 cycles	2MB / 16-way / 34 cycles
Skylake Client S	32kB / 8-way / 4 cycles	256kB / 8-way / 12 cycles	2MB / 16-way / 34 cycles
Skylake Client U	32kB / 8-way / 4 cycles	256kB / 8-way / 12 cycles	2MB / 16-way / 34 cycles
Skylake Client Y	32kB / 8-way / 4 cycles	256kB / 8-way / 12 cycles	8MB / 16-way / 34 cycles
Skylake Server DE	32kB / 8-way / 4 cycles	256kB / 8-way / 14 cycles	8MB / 16-way / 34 cycles
Skylake Server SP	32kB / 8-way / 4 cycles	256kB / 8-way / 14 cycles	8MB / 16-way / 34 cycles
Skylake Server W	32kB / 8-way / 4 cycles	256kB / 8-way / 14 cycles	8MB / 16-way / 34 cycles
Skylake Server X	32kB / 8-way / 4 cycles	256kB / 8-way / 14 cycles	8MB / 16-way / 34 cycles
Ice Lake	32kB / 8-way / 4 cycles	256kB / 8-way / 12 cycles	8MB / 16-way / 28 cycles
Artificial 1	32kB / 8-way / 4 cycles	256kB / 8-way / 12 cycles	8MB / 16-way / 34cycles
Artificial 2	64kB / 2-way / 4 cycles	512kB / 4-way / 12 cycles	No L3
Artificial 3	32kB / 2-way / 3 cycles	512kB / 16-way / 24 cycles	8MB / 32-way / 52 cycles
Artificial 4	64kB / 8-way / 3 cycles	1MB / 8-way / 20 cycles	32MB / 16-way / 28 cycles
Artificial 5	32kB / 4-way / 5 cycles	256kB / 4-way / 16 cycles	8MB / 32-way / 44 cycles
Artificial 6	64kB / 8-way / 4 cycles	1MB / 8-way / 16 cycles	8MB / 32-way / 36 cycles
Artificial 7	16kB / 8-way / 3 cycles	512kB / 16-way / 12 cycles	32MB / 32-way / 28 cycles
Artificial 8	32kB / 2-way / 2 cycles	1MB / 16-way / 16 cycles	32MB / 32-way / 52 cycles
Artificial 9	64kB / 4-way / 5 cycles	256kB / 4-way / 24 cycles	No L3

Table 3.12: Port organization of the architectures used to evaluate the bug localization mechanisms.

Architecture	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6
Sandy Bridge	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Divider	1 ALU, 1 Vector Unit, 1 Int Mult, 1 FP Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit, 1 Branch Unit, 1 FP Unit	-
Ivy Bridge	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Divider	1 ALU, 1 Vector Unit, 1 Int Mult, 1 FP Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit, 1 Branch Unit, 1 FP Unit	-
Haswell	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit
Broadwell	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit

Table 3.12: Port organization of the architectures used to evaluate the bug localization mechanisms.

Architecture	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6
Skylake Client DT	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit
Skylake Client H	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit
Skylake Client S	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit

Table 3.12: Port organization of the architectures used to evaluate the bug localization mechanisms.

Architecture	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6
Skylake Client U	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit
Skylake Client Y	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit
Skylake Server DE	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit

Table 3.12: Port organization of the architectures used to evaluate the bug localization mechanisms.

Architecture	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6
Skylake Server SP	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit
Skylake Server W	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit
Skylake Server X	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit

Table 3.12: Port organization of the architectures used to evaluate the bug localization mechanisms.

Architecture	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6
Icelake	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit
Artificial 1	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit
Artificial 2	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit

Table 3.12: Port organization of the architectures used to evaluate the bug localization mechanisms.

Architecture	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6
Artificial 3	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit
Artificial 4	1 ALU, 1 FP Unit, 1 Int Vector, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 Int Unit	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Branch Unit
Artificial 5	1 ALU, 1 Vector Unit, 1 Int Mult	1 ALU, 1 Vector Unit	1 ALU, 1 Vector Unit	1 Load Unit	1 Store Unit	1 FP Unit, 1 FP Unit	1 ALU, 1 Branch Unit
Artificial 6	1 ALU, 1 Vector Unit, 1 FP Unit, 1 Int Mult, 1 Divider, 1 Branch Unit	1 ALU, 1 Vector Unit, 1 FP Mult, 1 FP Unit, 1 Int Mult	1 Load Unit	1 Load Unit	1 Store Unit	1 ALU, 1 Vector Unit	1 ALU, 1 Vector Unit

Table 3.12: Port organization of the architectures used to evaluate the bug localization mechanisms.

Architecture	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6
Artificial 7	1 Load Unit, 1 Store Unit	1 ALU, 1 Int Mult	1 ALU, 1 Branch Unit	1 FP Mult, 1 Divider	1 FP Unit	-	-
Artificial 8	1 ALU, 1 Vector Unit, 1 Int Mult	1 ALU, 1 Vector Unit	1 ALU, 1 Vector Unit	1 Load Unit	1 Store Unit	1 FP Unit, 1 FP Unit	1 ALU, 1 Branch Unit
Artificial 9	1 ALU, 1 Load Unit, 1 Store Unit, 1 Vector Unit, 1 Int Mult, 1 Divider	1 ALU, 1 Vector Unit, 1 FP Unit, 1 Branch Unit	1 Load Unit	1 Store Unit	-	-	-

3.5.7.3 Implemented bugs

As for the case of bug detection, the gem5 [33] simulator is considered as performance bug-free given its wide acceptance on the computer architecture community. Therefore, in order to evaluate the methodologies for performance bug localization, performance bugs are artificially injected. Some of the bugs used to evaluate this technique were also utilized for the evaluation of the performance bug detection methodology, while others are new for this evaluation.

The localization techniques are examined under two unit granularities, **coarse** (composed by 11 units) and **detailed** (composed by 16 units). The assignment of the bug locations on each granularity and the description of each performance bug can be found in Table 3.13. Since the evaluation of these techniques is based on gem5 simulations, no further breakdown is possible, as most of the physical structures of each unit are implemented as an abstracted data structure in gem5.

Table 3.13: Performance bug types injected to gem5 and their corresponding locations for both granularities.

Bug location per granularity		**Bug Description**
Coarse	**Detailed**	
Fetch	Fetch	Fetching instructions from the instruction cache takes "T" cycles longer than expected.
		Every "T" cycles, the maximum number of instructions that the processor is able to fetch is reduced by "N" entries during one cycle.
Decode	Decode	Instructions that require no source operands are delayed by "T" cycles on the decode stage.
Issue	Instruction Scheduling	If an instruction with opcode "X" reaches the front of the instruction queue, meaning that it has become the oldest instruction there, then the issue process is stalled until the instruction can be issued (all the dependencies have been met, and computational resources are available). Once this occurs, only that instruction leaves the queue during that cycle. Normal behavior is resumed afterward.

Table 3.13: Performance bug types injected to gem5 and their corresponding locations for both granularities.

Bug location per granularity		**Bug Description**
Coarse	**Detailed**	
		Every instruction whose opcode is "X" can be retired from the instruction queue only when it becomes the oldest instruction there. A similar bug was found in the Intel Xeon Processors errata [76]. Its description can be found under "POPCNT instruction may take longer to execute than expected".
	Dependency Solver	If the operands of instructions with opcode "X" depend on the result of an instruction with opcode "Y", the issuing of the former is stalled by "T" cycles after its operands are ready.
	Instruction Queue	If less than "N" slots are available in the instruction queue, delay the next instruction by "T" cycles.
		The pointer signaling the front of the instruction queue is randomly shifted by "N" positions. This event occurs with a frequency of "T" times per 1000 cycles.
Rename	Rename	All instructions whose opcode is "X" are marked as serializing instructions. This causes all subsequent instructions to be stalled until that instruction has been issued.
Execute	Functional Units	The latency of functional units handling integer operation "X" is increased by "T" cycles.
		The latency of functional units handling floating-point operation "X" is increased by "T" cycles.
		The latency of functional units handling "Single-Instruction Multiple-Data" operations "X" is increased by "T" cycles.
Branch	Branch	Branch prediction index table malfunction, effectively reducing the size of the tables by "N" entries.
Registers	Register Delay	If an instruction with opcode "X" uses physical register "R", then this instruction is delayed by "T" cycles. A bug similar to this can be found on Intel 386 DX errata [79] labeled as "POPA/POPAD instruction malfunction".

Table 3.13: Performance bug types injected to gem5 and their corresponding locations for both granularities.

Bug location per granularity		**Bug Description**
Coarse	**Detailed**	
		After every "N" times a register has been written, delay the following write by "T" cycles. The inspiration for this bug is the one labeled as "GPMC may stall after 256 write accesses in NAND_DATA, NAND_COMMAND, or NAND_ADDRESS" found on the TI AM3517 and TI AM3505 ARM processors errata [78].
	Register Count	The number of physical registers is reduced by "N".
Load/Store Queue	Load Queue	For every "N" requests, the load-queue incorrectly rejects entries stating that it is full.
	Store Queue	For every "N" requests, the store-queue incorrectly rejects entries stating that it is full.
Memory	Memory Operations	After every "N" stores to the same cache line, delay the following write by "T" cycles.
	L2 Cache Latency	The latency of L2 cache is "T" cycles higher than expected. This issue is similar to the documented for NXP MPC7448 RISC processor in its errata [77] labeled as "L2 latency perfomance issue".
Re-Order Buffer	Re-Order Buffer	If less than "N" slots are available in the re-order buffer, delay the next instruction by "T" cycles.
Commit	Commit	Every "T" cycles, the maximum number of instructions that the processor is able to commit is reduced by "N" entries during one cycle.

For each of these bug types, at least three variations were implemented by varying the values of "X", "Y", "N", "R", and "T". These bugs are divided into four different categories, according to their average IPC degradation (measured across the 10 SPEC CPU2006 applications used). The distribution of the impact of these bugs is shown in Figure 3.20. In this case, "High" impact means an average degradation $\geq$5%. These bugs are usually easier to debug due to their high impact, therefore, only a small fraction of the implemented bugs is in this bin. The "Moderate" impact bins covers degradation between 5%$-$1%, while the "Low" covers cases between 1%$-$0.1%. Finally,

in the "Very Low" bin are the bugs with average IPC impact <0.1%. Although the average IPC impact of the bugs in the "Very Low" bin is so small that it might not be a priority to fix on tight product release schedules, they are analyzed as a stress-test of the proposed techniques. It is also possible that a low impact bug on the evaluated traces could produce significant impact on other workloads. More importantly, a performance bug, even a small one, may have subtle implications to other issues of the design, and a clean design will always be preferred.

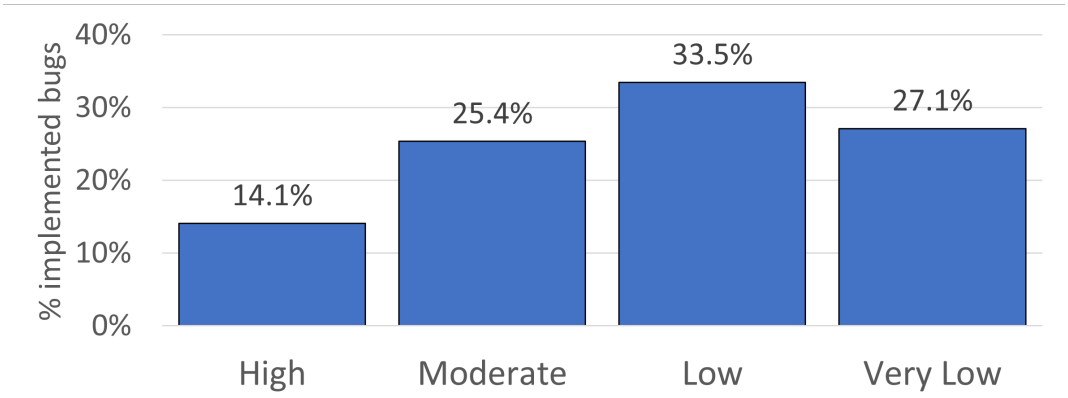


Figure 3.20: Average IPC impact distribution of bugs injected for performance bug localization.

3.5.7.4 Training and test data organization for the bug localization task

The set of injected bugs is partitioned into two disjoint groups: the "seen" and the "unseen" bugs. At least one bug variant from every type is placed in the "unseen" set. This separation is done to evaluate the methodologies on bug variations which have not been used for training. The data organization is as follows:

- Training data:

    1. Data with positive labels: For each model trained to identify performance bugs in unit u_j, the samples with positive labels correspond to those from bugs in unit u_j which are placed in the "seen" category. These include data exclusively from the "train" microarchitectures.

2. Data with negative labels: For each model corresponding to unit u_j, the samples with negative labels are those from bug-free cases or bugs that do not occur in unit u_j and are placed in the "seen" category. These include data exclusively from the "train" microarchitectures.

- Testing data:

1. Designs with bugs: Samples from both "seen" and "unseen" bugs exclusively from "test" architectures are evaluated. In any given sample, only one bug is inserted.

2. Bug-free designs: As mentioned earlier, it is assumed that a performance bug has already been detected on the design under test, either by manual methods or by using the methodology detailed in Section 3.4. However, an analysis of the behavior of bug-free architectures using the methodologies is shown in Section 3.5.8.6 to evaluate when false-positives at the detection step occur.

It is important to note that all the samples used for training are taken from "seen" bugs in "train" architectures. On the other hand, every sample used for testing comes from "test" architectures but can either be a "seen" or an "unseen" bug.

3.5.8 Bug localization results

This section presents the details of the experiments conducted to evaluate the proposed methodologies.

Since the localization problem is formulated as a multi-class classification task, both methodologies produce a sorted list from highest to lowest probability of the performance bug being located in each unit. The top-*k* accuracy metric is used to measure results quality. Here, a result is considered correct if the actual bug location is found among the first *k* choices suggested by our techniques. The reason why top-*k* accuracy is more relevant in this work than other metrics (*e.g.* a confusion matrix) is because, rather than measuring when a sample is incorrectly classified into a different class, it is more important to determine how high in the predicted ranking the actual bug

location is, as this represents how many units the designer would have to search to actually find the bug.

Note that the accuracy of a random guess in a multi-class task is not 50%, as in the more frequently studied binary classification, instead it is actually $1/|U|$ where U is the set of all classes (units). Therefore, the top-*k* accuracy of a random guess is given by $k/|U|$. The reported accuracy is obtained as the average across the four architectures used as test cases.

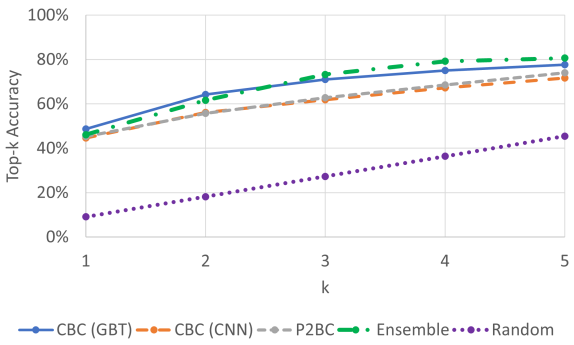
3.5.8.1 Bug localization overall accuracy

In this section, the top-*k* accuracy across the implemented bugs in the test architectures is shown. Figures 3.21a and 3.21b show how the top-*k* accuracy behaves as the value of *k* increases across all the implemented bugs ("seen" and "unseen"). While Figures 3.21c and 3.21d show the same, but for the "unseen" bugs exclusively. In each figure, the results for different implementations of CBC, P2BC and ensemble methodologies are shown for both granularity levels. The figures show values of *k* less than 5 as longer lists of possible locations would not provide significant help to designers to conduct their debug.

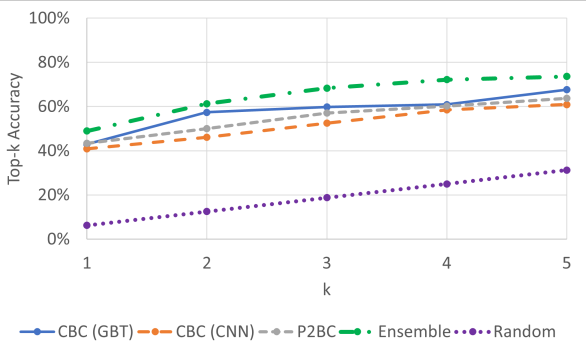
In the figures, "CBC (GBT)" refers to the case of CBC with per-time-step classification and gradient boosted tree models used, while "CBC (CNN)" performs a per-trace classification using convolutional neural networks, as described in Section 3.5.3. The "P2BC" results correspond to an implementation with CNNs for the classifiers on the second stage. The first stage uses gradient boosted trees models for IPC estimation, as it was shown to be the best performing in Section 3.4.6. The results labeled as "Random" represent what would be obtained by using a random guess.

It can be seen from the figures that results obtained across all bugs (Figures 3.21a and 3.21b) do not differ significantly from the results obtained for "unseen" bugs only (Figures 3.21c and 3.21d) in any method, demonstrating that the methodologies are robust to handle new bugs that were not used to trained the ML models. Another finding is that, as expected, using a coarser granularity achieves a higher accuracy than a more detailed one.

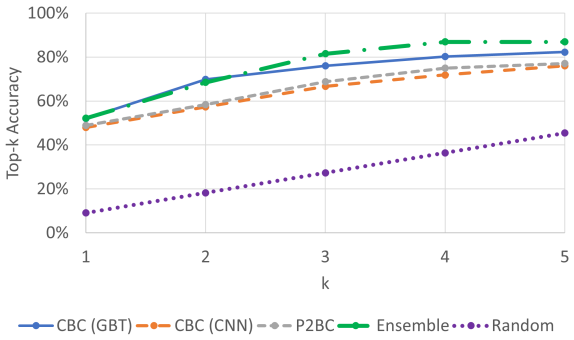
When each technique is used individually, it was found that "CBC (GBT)" is the best performing for both coarse and detailed granularities, being able to identify the correct location of the bug



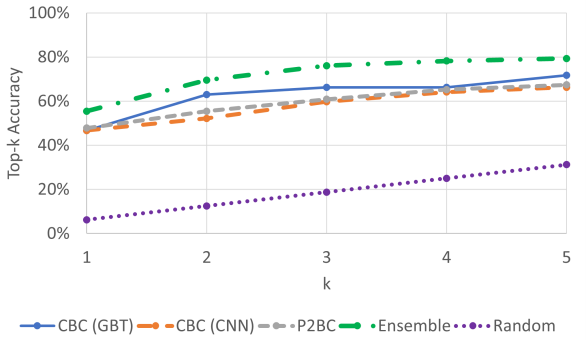
(a) Coarse class granularity, all evaluated bugs.



(b) Detailed class granularity, all evaluated bugs.



(c) Coarse class granularity, only "unseen" bugs.



(d) Detailed class granularity, only "unseen" bugs.

Figure 3.21: Top-*k* bug localization accuracy for different methodologies.

in the top-3 ranked options in more than 70% of the cases with coarse granularity and 60% of the cases with detailed granularity. Although the accuracy achieved by P2BC is not as high as the obtained by CBC, it provides satisfactory results while using $100\times$ less storage.

The ensemble technique, which combines the "CBC (GBT)" and the "P2BC" implementations, outperforms each individual method. This is even more pronounced on the detailed granularity, gaining a 10% increase in top-3 accuracy.

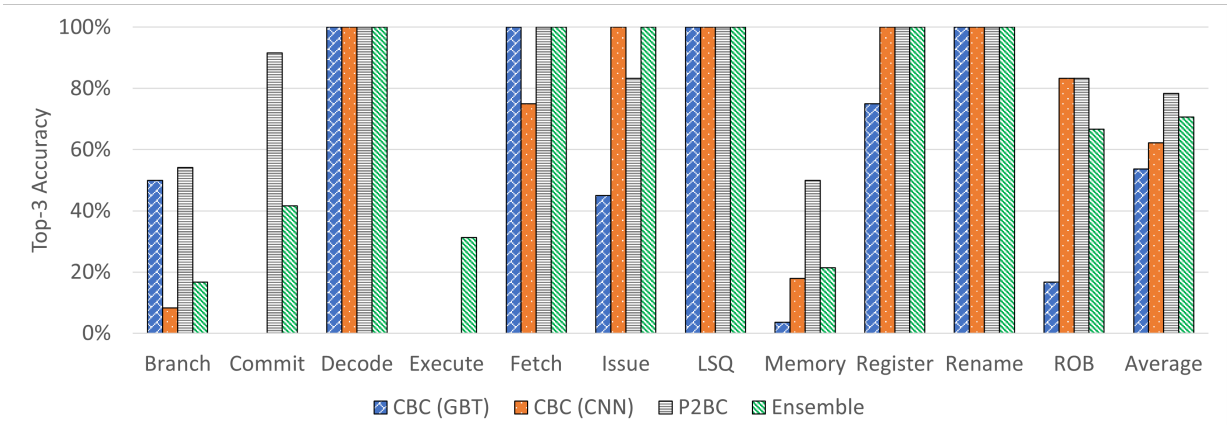
Although the overall accuracy across the implemented bugs is very similar in the evaluated methodologies, there is a significant difference on how well they identify each individual unit. An analysis of how well each methodology performs across the different units is presented in Section 3.5.8.2.

The experiments shown in this section put together all IPC impact bins, including bugs with an average IPC impact $<$0.1%. A detailed study of the behavior on bugs with higher impact is

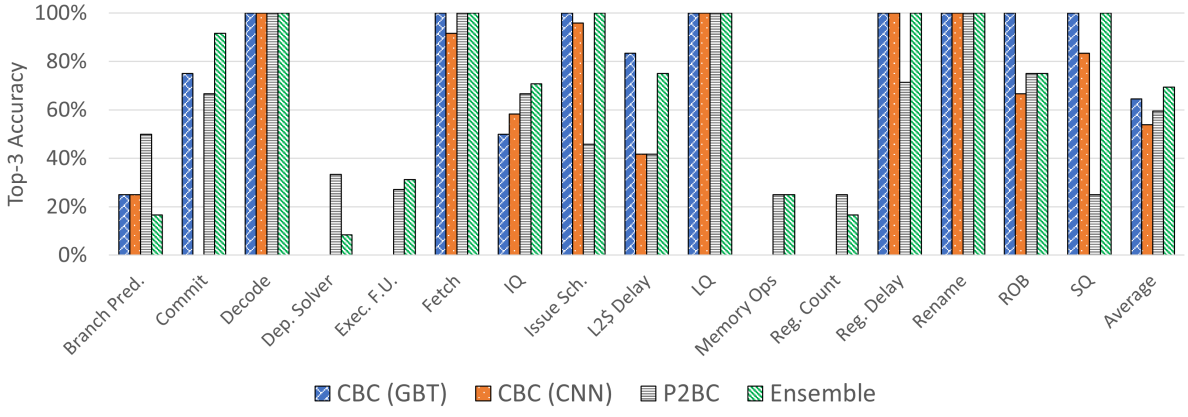
presented in Section 3.5.8.3.

3.5.8.2 Bug localization accuracy per microarchitectural unit

In this section, the accuracy of the methodologies is partitioned by classes in order to demonstrate the quality of results for each unit. In Figure 3.22 the top-3 accuracy obtained for each unit for coarse and detailed granularity is shown, as well as the average top-3 accuracy across all labels. Since the implemented bugs are not uniformly distributed across all the possible labels, this average accuracy differs slightly from the presented in Section 3.5.8.1.



(a) Coarse label granularity.



(b) Detailed label granularity.

Figure 3.22: Top-3 accuracy for each possible label for different methodologies across all bugs.

As it can be observed from the figures, there are some classes on which the achieved accuracy is very low, such as "Execute" and "Memory" in the coarse granularity. This problem is more pronounced in CBC, although P2BC is not exempt from it. The results for the ensemble methodology show that, by merging the information of both models, the deficiencies on some units can be attenuated with small impact on the accuracy of others.

In the case of the detailed unit granularity, the major issue appears on the units that were broken down from the coarser granularity *e.g.* all the bugs in "Register" unit for the coarse granularity are divided into "Register Count" and "Register Delay" for the detailed one. In this case, the majority of bugs incorrectly localized for the label "Register Count" have "Register Delay" as one of the top-3 options, and although this is still an inaccurate result, it provides the designers with a possible location that is related to the actual bug.

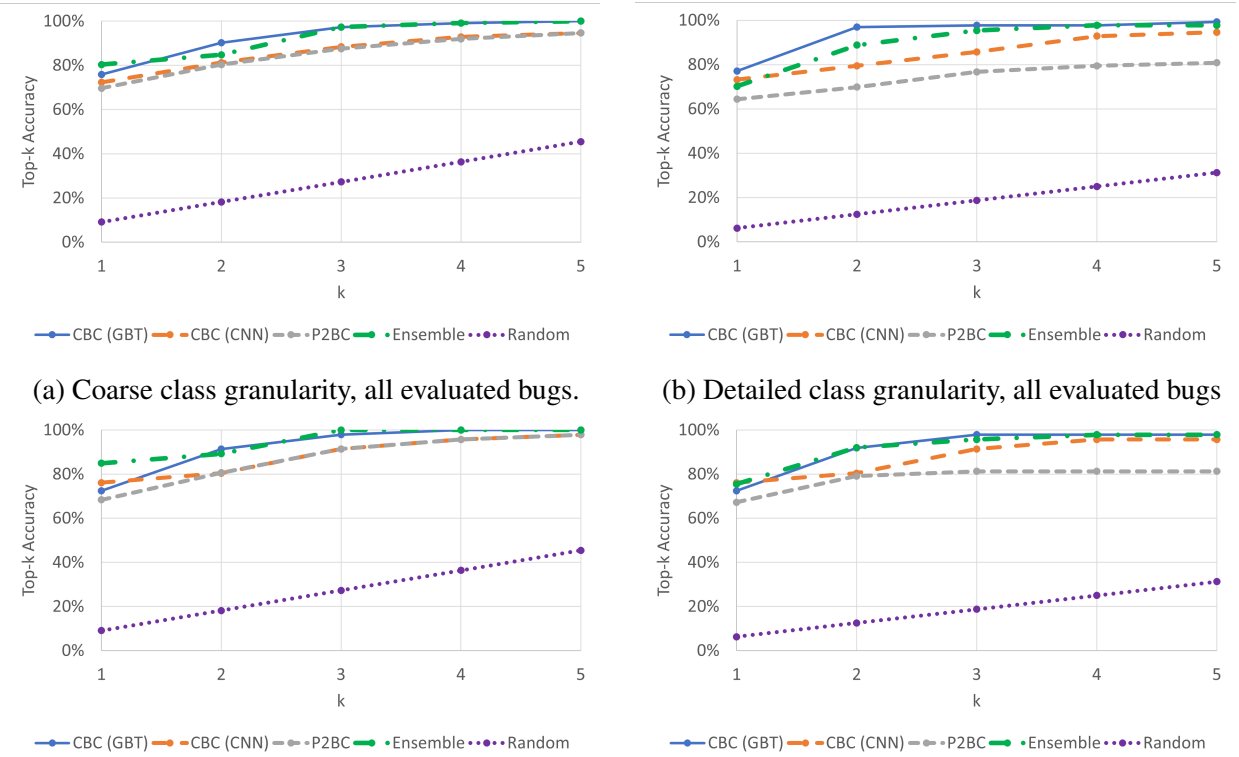
For the detailed granularity, the gain of using the ensemble is even larger than on the coarse granularity. In this case, the top-3 accuracy achieved by the ensemble method is at least $2.5\times$ larger than a random guess in every label, except for the "Dependency Solver" unit. The ensemble methodology achieves an accuracy greater or equal to the best performing individual method in 63% of the units in the coarse granularity and 68% for the detailed one, making it a more balanced choice across the different units.

One important note for this analysis is that the majority of bugs in the classes with this issue have an average IPC impact below 0.1%. This not only makes the localization task more difficult, but also reduces the importance of the inaccuracy, due to the small impact on the overall quality of the design.

3.5.8.3 Bug localization accuracy per impact bin

As shown in Section 3.5.8.1, the methodologies proposed here have over $\sim 60\%$ top-3 accuracy across all the bugs studied in this work. It is important to note, however, that accurate localization of some bugs is more important than others. For example, bugs with average IPC impact lower than 0.1% are not typically a priority for most designs, since the gain by fixing them is not significant. For that reason, in this section the top-*k* accuracy for different levels of IPC degradation is analyzed.

Results shown in Figure 3.23 show the behavior of the top-k accuracy as k increases in the case with average IPC impact greater than 1%.



(a) Coarse class granularity, all evaluated bugs.

(b) Detailed class granularity, all evaluated bugs

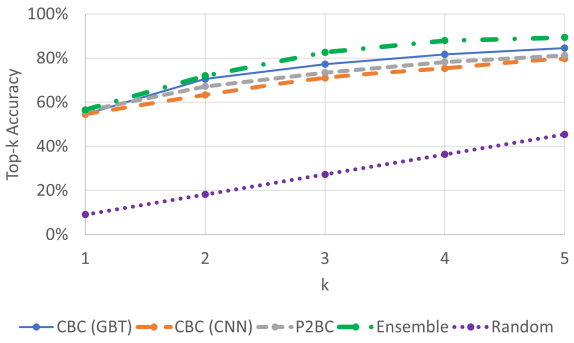
(c) Coarse class granularity, only "unseen" bugs.

(d) Detailed class granularity, only "unseen" bugs.

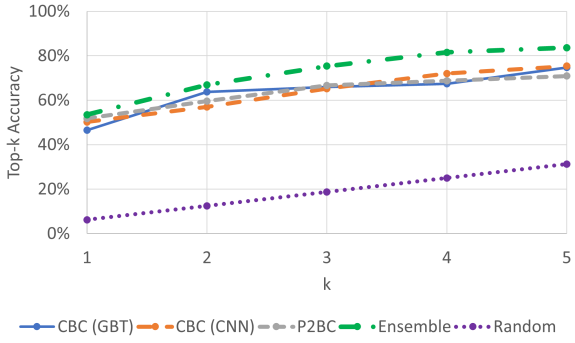
Figure 3.23: Top-k localization accuracy for different methodologies in bugs with average IPC impact $> 1\%$.

Results show that a top-3 accuracy greater than 98% is obtained for bugs with an average IPC impact greater than 1% for both granularities, even on the case of "unseen" bugs. In both granularities the results obtained by CBC are better than those of P2BC, but the ensemble technique achieves results almost as good as the obtained by CBC in the detailed granularity and even better on the coarse one.

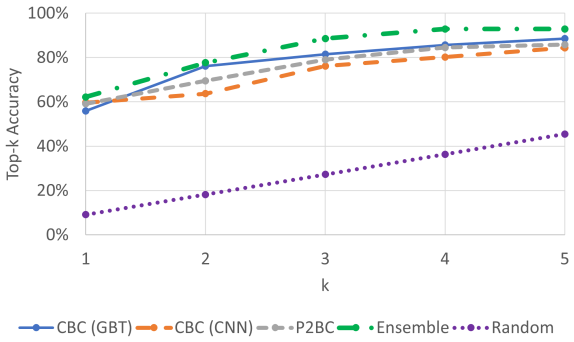
Figure 3.24 shows the results when the bugs with an average IPC impact $> 0.1\%$ are considered. Figures 3.24a and 3.24b show the overall results across all bugs, while Figures 3.23c and 3.24d shows the results exclusively on the "unseen" bugs.



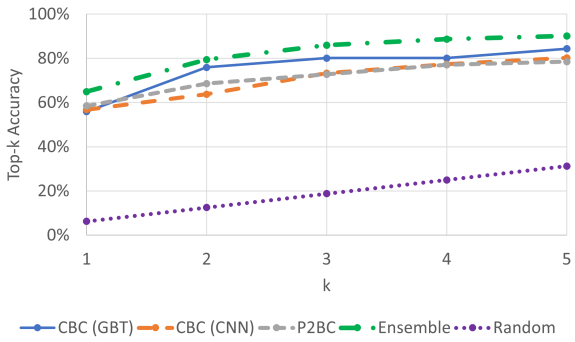
(a) Coarse class granularity, all evaluated bugs.



(b) Detailed class granularity, all evaluated bugs



(c) Coarse class granularity, only "unseen" bugs.



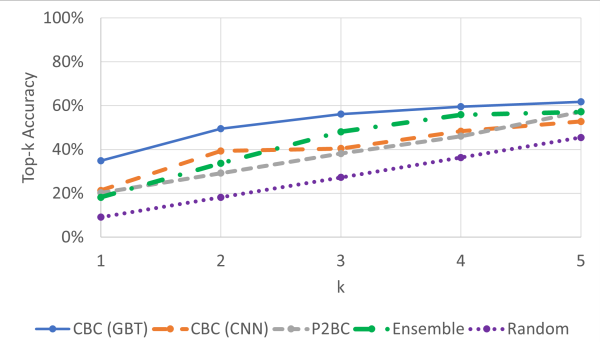
(d) Detailed class granularity, only "unseen" bugs

Figure 3.24: Top-*k* localization accuracy for different methodologies in bugs with average IPC impact $> 0.1\%$

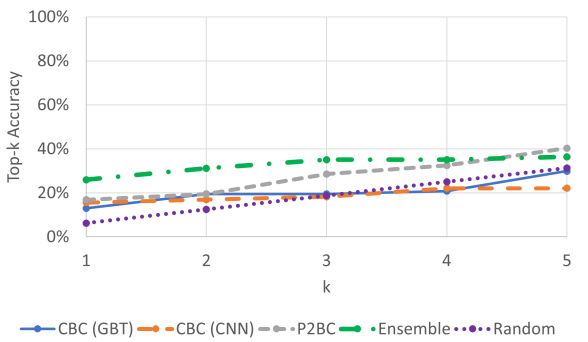
For bugs with average IPC impact greater than 0.1% the top-3 accuracy is still very high, at around 75% for the detailed granularity and over 82% on the coarse one across all the bugs, around $4\times$ what a random guess would obtain. The accuracy gap between both methodologies is reduced for smaller impact bugs, due to P2BC outperforming CBC on bugs in the "Low" average IPC impact bin. In this case, the advantage of using the ensemble technique is more prominent, as this method clearly outperforms both CBC and P2BC in every case.

Finally, Figure 3.25 shows the results when the bugs with an average IPC impact $< 0.1\%$ are considered. Figures 3.25a and 3.25b show the overall results across all bugs, while Figures 3.25c and 3.25d show the results exclusively on the "unseen" bugs.

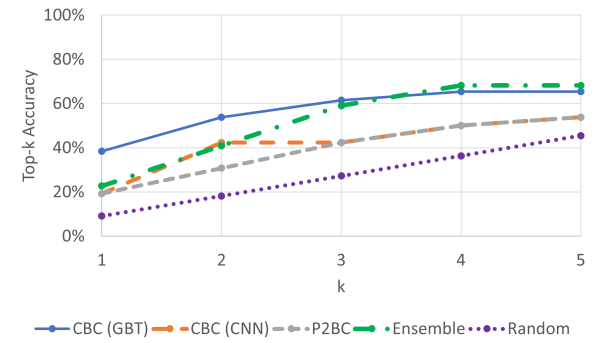
Even for these bugs with"Very Low" average IPC impact, the proposed methodologies achieve top-3 accuracy of around 55% on the coarse granularity, and up to 35% on the detailed across all



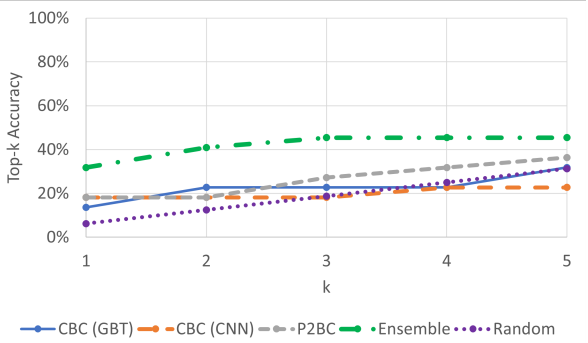
(a) Coarse class granularity, all evaluated bugs.



(b) Detailed class granularity, all evaluated bugs.



(c) Coarse class granularity, only "unseen" bugs.



(d) Detailed class granularity, only "unseen" bugs.

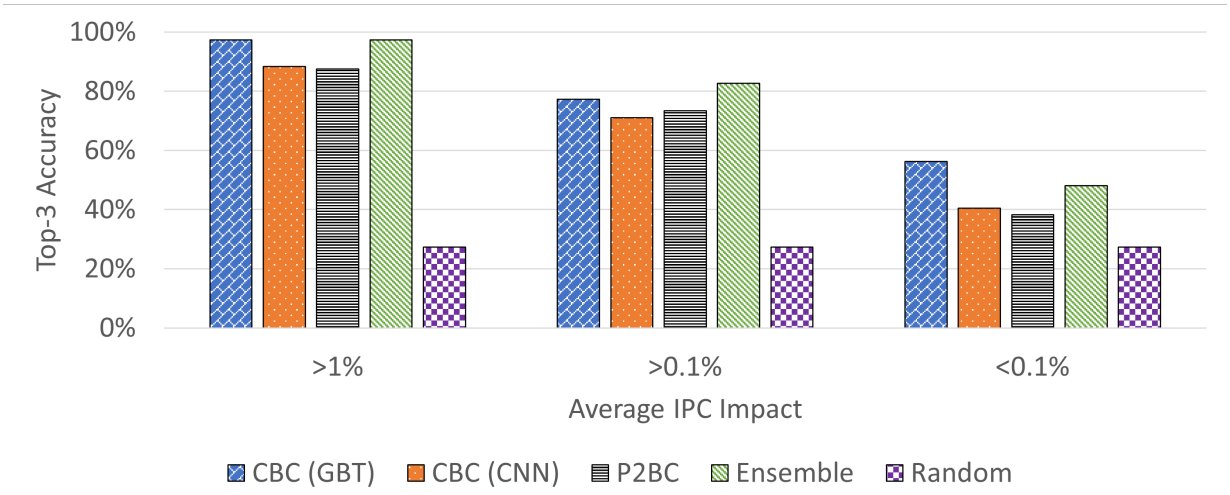
Figure 3.25: Top-k localization accuracy for different methodologies in bugs with average IPC impact $< 0.1\%$

the bugs evaluated. Thus, our methodologies perform decent location inferences, even on cases of performance bugs with minimal IPC impact on the evaluated workloads. The ensemble method achieves up to a 23% top-3 accuracy increase on bugs with average IPC impact smaller than 0.1% on the "unseen bugs".

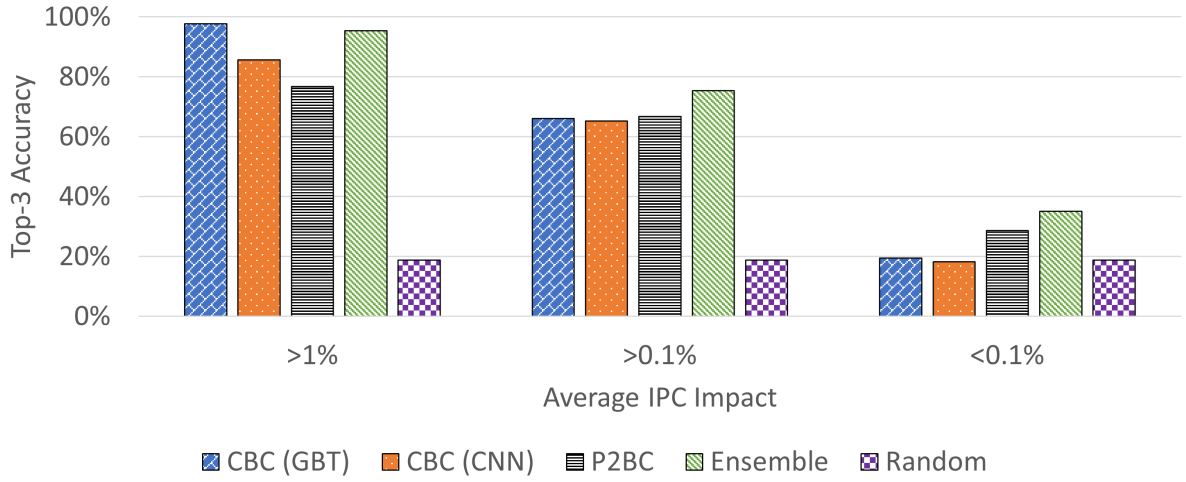
From this section, it can be concluded that the average IPC impact is a very significant factor for the accuracy of the methodologies, as the obtained accuracy decreases for bugs with small impact. This can be observed from the results shown in Figure 3.26 where the top-3 accuracy across all the evaluated bugs can be seen in both granularities partitioned by the average IPC impact produced by the bug.

Although high accuracy on every bug is highly desirable, higher impact bugs, on which our methods are very accurate, are far more important to localize than bugs with very small impact, on

which our methodologies only perform decently.



(a) Coarse label granularity.



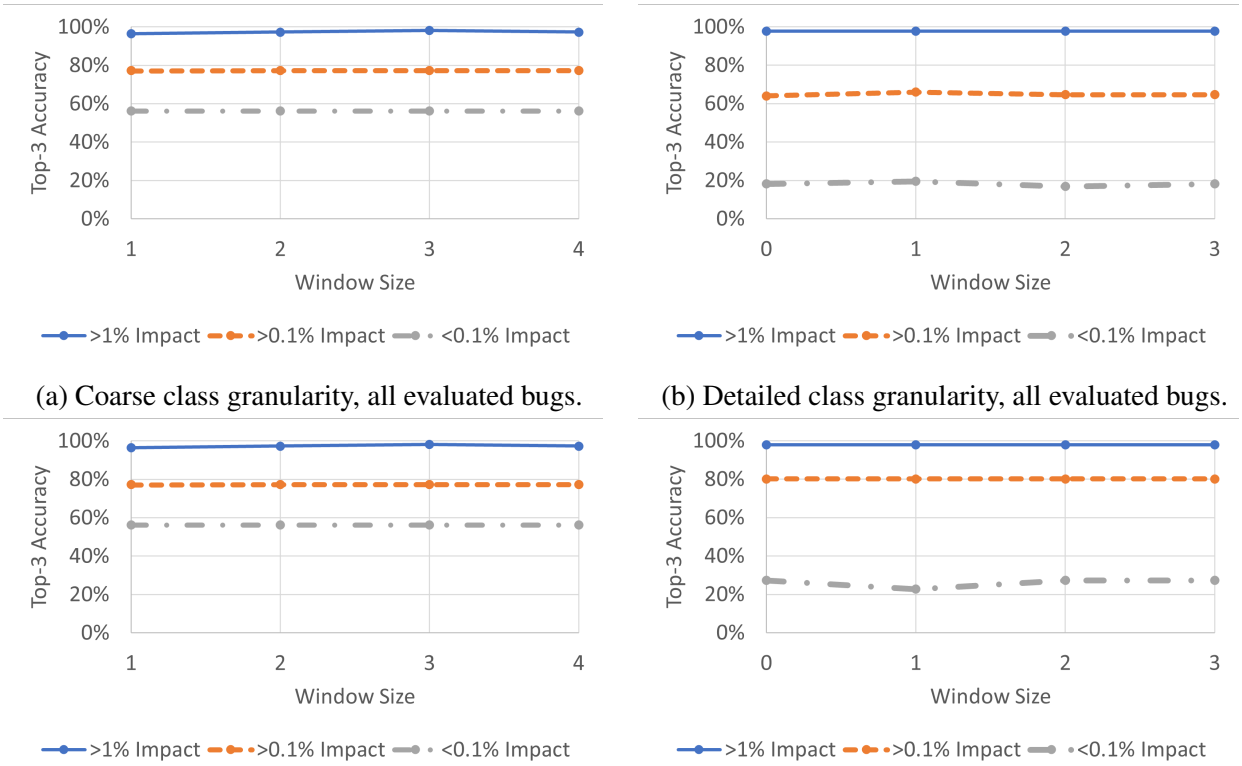
(b) Detailed label granularity.

Figure 3.26: Top-3 localization accuracy on different IPC impact level bugs for different methodologies considering all then evaluated bugs.

3.5.8.4 Impact of the window size on bug localization

As mentioned in Section 3.5.3, for the CBC methodology with per-time-step classification, the input features to the model can either be only the current time-step (*i.e.* a window of size 1) or

the current time-step accompanied by the features from $W - 1$ previous steps. In this section, an evaluation of how much impact the history window size has on model accuracy is conducted. Top-3 accuracy results for different levels of average IPC impact using the coarse unit granularity are shown in Figure 3.27.



(a) Coarse class granularity, all evaluated bugs.

(b) Detailed class granularity, all evaluated bugs.

(c) Coarse class granularity, only "unseen" bugs.

(d) Detailed class granularity, only "unseen" bugs.

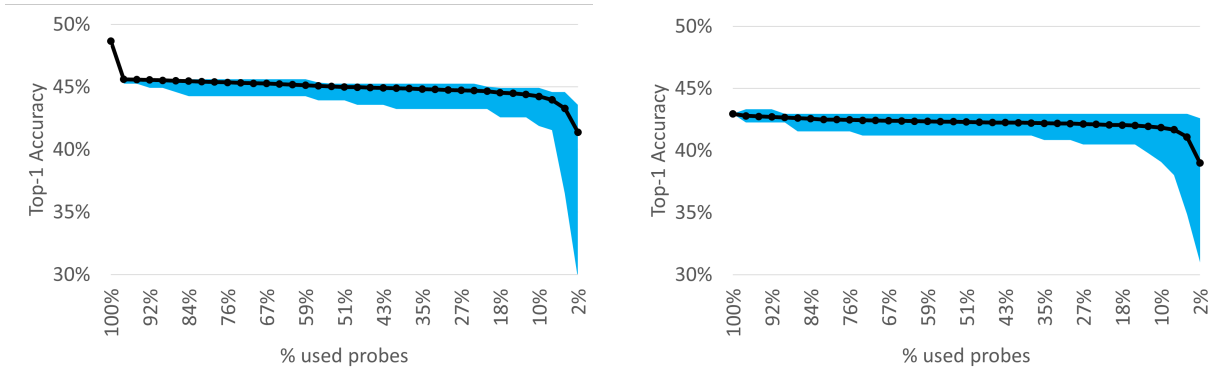
Figure 3.27: Impact of window size on top-3 localization accuracy.

As it can be observed from the figures, changes in the window size produced only minimal changes across the achieved top-3 accuracy for any average IPC impact bin. This behavior is likely due to the size of the time-steps. The 100k cycles sampling rate used here is large enough to encapsulate enough information about the behavior of the performance in the current time-step, making the information obtained by a history window irrelevant, and therefore, the size of the history window that is used as input features have essentially no impact on the accuracy of the

methodology.

3.5.8.5 Impact of the number of probes on bug localization

In this section, the impact of the number of probes on the overall accuracy is evaluated. This evaluation was conducted for the "CBC (GBT)" implementation, as it was found to be the best performing overall in previous evaluations. The experiment starts by using all 190 probes available for this work and the top-1 accuracy is measured. The number of used probes is iteratively reduced by randomly choosing five probes to be discarded in each iteration. The iterations continue until only five probes remain. This experiment is repeated 100 times to reduce the impact of the random choices of probe deletion and obtain a reliable trend. Figure 3.28 shows the obtained results.



(a) Coarse label granularity

(b) Detailed label granularity

Figure 3.28: Impact of the number of available probes on the top-1 localization accuracy across all evaluated bugs.

In the figures, the black line represents the average top-1 accuracy across the 100 iterations of the experiment, while the maximum and minimum accuracy of any individual iteration are represented by the shaded area. It can be observed from the figures that, as the number of probes used to test the methodology decreases, so does the average quality of results, making a point that as more probes are available, higher accuracy will be achieved. However, it can also be noted that the degradation in the overall accuracy is slow and proves that the methodology would be able to

achieve satisfactory results even with a reduced number of probes.

3.5.8.6 Handling of bug-free cases

As mentioned in previous sections, the evaluation of this work assumes that the presence of a performance bug has been detected. However, detection methodologies may still produce false-positives *i.e.* cases where a bug-free design is classified as buggy. In this section, an analysis of the behavior of both methodologies when such case occurs is conducted.

3.5.8.6.1 *Counter Based Classification (CBC)*

Each of the four "test" architectures without performance bugs produce a different ranking of units. However, the confidence assigned to the highest ranked unit is $\sim 5\times$ smaller than the average confidence obtained by the top-1 option in the case of architectures with bugs.

Although low confidence across all the units may be a signal of a false positive, it could also indicate a gap of coverage, as there might be a bug in a unit that is not considered in the evaluated classes. To prevent that, a model to detect "Bug-Free" architectures is trained in the same way the models for each possible bug location unit are trained (*i.e.* a "Bug-Free" class is included to the list of possible bug locations). Results show that such model performs very well, as it provides the highest ranked confidence score to bug-free identifications in all four bug-free "test" architectures. Including this class does not degrade localization results of buggy samples, as the confidence score for the "Bug-free" class is more than $200\times$ smaller than any 1st choice of all the buggy samples and it is not included as a top-5 option in any sample with a performance bug.

3.5.8.6.2 *Performance Prediction error-Based Classification (P2BC)*

For P2BC, the four "test" cases behaved in one of the following ways:

- High confidence from the model signaling "ROB" performance bugs, but very low confidence for every other unit ($< 0.1\%$). One possible explanation for this phenomenon is that ROB is the unit that "lies the closest" to the target metric (IPC), and therefore most performance counters of earlier stages are barely altered in the cases where a bug localized there occurs. Because of this, the IPC inference errors obtained by the first stage of this

methodology are very similar between "Bug-Free" and "ROB" cases.

- Multiple units with high confidence are obtained and there is nothing particular that differentiates these samples from the average behavior obtained in buggy samples. The two architectures that behave this way are the same two with higher IPC inference error in stage one of the methodology. Making a case that accurate IPC estimation is important for the correct functioning of this methodology.

Similarly to what was evaluated for CBC, a "Bug-Free" class addition was tested. In this case however, the method does not provide a satisfactory accuracy, making CBC more robust to handle false positive cases. One possible explanation for this is that, as shown in Section 3.5.8.3, P2BC is more sensible to bugs with small average IPC impact, which also makes it more vulnerable to false positives.

3.5.8.7 Potential debugging time speedup

In this section, the potential speedup that can be achieved when the proposed methodologies are used is evaluated. Here, the proposed methodologies are compared vs. a worst-case scenario of designers using a random guess to localize the bugs. It must be noted that although the proposed methodologies can perform a bug localization inference in only a few minutes, the produced output is a list of the most likely bug locations, designers must still conduct a manual debug to confirm that a bug is actually present in the suggested location.

Without loss of generality, a uniform time of $T_{local-debug}$ is assumed, this represents the time that would take a designer to debug one possible bug location, irrespective of the bug type, in order to confirm whether the bug is there or not. The values of $T_{local-debug}$ can be sometimes in the order of a few days or even weeks.

These assumptions may not be entirely realistic, as debuggers or architects may make an educated guess on bug location which could (or not) be better than random. In addition, the time to debug different locations may vary depending on their complexity and the familiarity of the designer with it. Nevertheless, this section provides an interesting evaluation of the potential de-

bugging speedups that can be achieved by using the proposed methods.

The average time it takes to confirm the localization of a bug can be obtained by adding the time required to debug the first *k* localization choices weighted by the probability of obtaining the correct class at the *k*-th guess. This can be generalized as:

$$\overline{T}_{total} = \sum_{k=1}^{k=|U|} k\, T_{local-debug}\, P(\hat{u}_k = u^*) \tag{3.16}$$

where $\overline{T}_{total}$ is the average time required to confirm the correct localization of a performance bug using the locations in the set U. The *k*-th ranked option provided by the method is denoted by $\hat{u}_k$, while the actual bug location is u^*. The probability of finding the correct location at the *k*-th ranked option is denoted as $P(\hat{u}_k = u^*)$, and can be approximated by the accuracy obtained on the *k*-th choice using the evaluated bugs.

For example, if the results from the "CBC (GBT)" for bugs with an average IPC impact greater than 1% are used, around 77% of the time the method provides the correct location on the first option, meaning that it only takes designers $T_{local-debug}$ to confirm the location of the bug. If the first choice is incorrect, then in addition to the time spent on the first choice, another $T_{local-debug}$ must be spent confirming if the bug is in the second location. This means that around 20% of the time, the effort to confirm the bug location takes $2T_{local-debug}$. By continuing this exercise, it can be computed that the average time to confirm the correct bug location using the "CBC (GBT)" methodology is $1.3T_{local-debug}$. By comparison, the time needed using a random guess would be $8.5T_{local-debug}$, which means a speedup of about $6.5\times$. The results of speedup gained by the proposed methodologies vs. a random guess are shown in Figure 3.29.

Please note here that the top-*k* accuracy is a cumulative metric, which means that it includes the cases where the right choice was found on either of the first *k* choices, while $P(\hat{u}_k = u^*)$ requires the accuracy at the *k*-th choice exclusively.

The results show that the proposed methodologies achieve a localization speedup of up to $\sim 6.5\times$ the time estimated using random guess for bugs with average IPC impact greater than 1% and up to $\sim 2.5\times$ when all the bugs with average impact greater than 0.1% are considered. This is

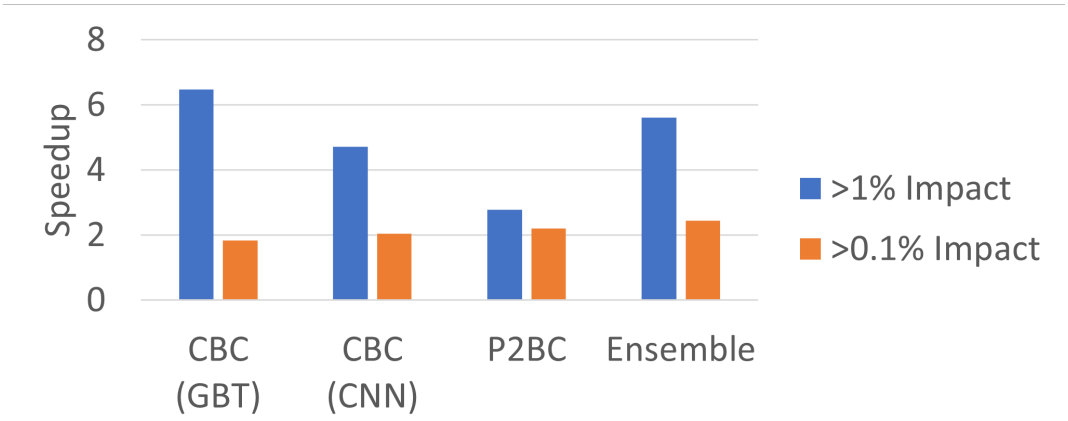


Figure 3.29: Speedup gained in debugging time by using the proposed bug localization methodologies.

a very significant gain, especially considering the large values $T_{local-debug}$ can take sometimes.

Overall, both methodologies show great accuracy, which translates in very significant speedups on the bug localization process. CBC achieved great accuracy for bugs with significant average IPC impact, which makes it a good choice for bug localization when high IPC regressions are observed, while P2BC proved to be more sensible to smaller impact bugs with $100\times$ smaller footprint, which makes it a more feasible choice on infrastructures with less storage available, or for debugging bugs with small average IPC impact.

3.6 Conclusion

In this chapter, we presented the first systematic studies on the usage of machine learning for performance bug detection and localization in microprocessor cores, to the best of our knowledge. The machine learning models extract knowledge from legacy designs and avoid the need for reference performance models used in previous works, which are error prone and time consuming to construct. Further, we leverage the usage of SimPoints [73] as microbenchmarks that can be automatically extracted in order to obtain short, relevant and performance orthogonal traces for performance probing. Simulation results show that our methodology can detect 91.5% of the bugs with impact greater than 1% of the IPC on a new microarchitecture when completely new bugs

exist in a new microarchitecture. As far as localization, our simulation results on the evaluated bugs show that the top-3 accuracy achieved is as high as 98% for bugs with average IPC impact greater than 1%, and 75% for bugs with impact greater than 0.1%.

4. CONCLUSION

In this dissertation, we propose machine learning-based techniques to tackle two different problems on the IC design industry: pre-routing timing estimation, and performance debug.

The pre-routing timing estimation technique achieved high estimation accuracy on post-route timing while using only the data extracted from a placed, but unrouted circuit database. This helps designers to have accurate timing estimations to apply in several optimizations that need to be executed before running routing, while preventing over-designing due to pessimistic estimations. Our pre-route estimations proved to be as correlated with the sign-off timer as the commercial tool, but with a much lower estimation error. This was the first systematic study on the application of machine learning for pre-routing timing estimation, to the best of our knowledge.

The performance debug task was subdivided into two different sub-tasks:

1. **Bug detection**: Our proposed ML-based technique achieved a 91.5% detection accuracy while incurring in no false-positives when the evaluated bugs had an IPC impact greater than 1%, the overall detection rate achieved was of 84%, when bugs with IPC impact smaller than 1% are also considered.

2. **Bug localization**: Two ML-based techniques were proposed, evaluated and contrasted. Between the two approaches, one is more accurate while the other is $10\times$ faster and has $100\times$ smaller data footprint. The two approaches can be integrated to provide further accuracy improvement. Simulation results on the evaluated bugs show that the top-3 accuracy achieved is as high as 98% for bugs with average IPC impact greater than 1%, and 75% for bugs with impact greater than 0.1%.

These were the first studies on systematic performance debugging, to the best of our knowledge and, although multiple solutions were proposed and evaluated, the area of performance validation still has many more opportunities to be advance and we hope to draw the attention of the research community to this problem.

4.1 Future Work

For the pre-routing timing estimation, in future work we will develop more sophisticated feature selection techniques in order to further improve the prediction accuracy. We will also look for larger and more complicated test cases for stress test of the proposed techniques, e.g., cases with more metal layers, more routing congestion and more buffer insertions.

For the performance debugging, we plan to explore automated root cause analysis techniques for localized bugs. Although determining the location of the bug is extremely helpful, a methodology to provide detailed information regarding the mechanisms producing the bug and how to fix is still missing. We will also extend the detection and localization methodologies for multi-core memory systems and on-chip communication fabrics.

REFERENCES

[1] E. Carvajal Barboza, N. Shukla, Y. Chen, and J. Hu, "Machine learning-based pre-routing timing prediction with reduced pessimism," in *ACM/IEEE Design Automation Conference*, pp. 1–6, 2019.

[2] E. Carvajal Barboza, S. Jacob, M. Ketkar, M. Kishinevsky, P. Gratz, and J. Hu, "Automatic microprocessor performance bug detection," in *IEEE International Symposium on High-Performance Computer Architecture*, pp. 545–556, 2021.

[3] G. Huang, J. Hu, Y. He, J. Liu, M. Ma, Z. Shen, J. Wu, Y. Xu, H. Zhang, K. Zhong, X. Ning, Y. Ma, H. Yang, B. Yu, H. Yang, and Y. Wang, "Machine learning for electronic design automation: A survey," *ACM Transactions on Design Automation of Electronic Systems*, vol. 26, no. 5, 2021.

[4] R. Singhal, K. Venkatraman, E. R. Cohn, J. G. Holm, D. A. Koufaty, M.-J. Lin, M. J. Madhav, M. Mattwandel, N. Nidhi, J. D. Pearce, *et al.*, "Performance analysis and validation of the Intel® Pentium® 4 processor on 90nm technology.," *Intel Technology Journal*, vol. 8, no. 1, pp. 39–48, 2004.

[5] B. Black, A. S. Huang, M. H. Lipasti, and J. P. Shen, "Can trace-driven simulators accurately predict superscalar performance?," in *IEEE International Conference on Computer Design. VLSI in Computers and Processors*, pp. 478–485, 1996.

[6] R. Desikan, D. Burger, and S. W. Keckler, "Measuring experimental error in microprocessor simulation," in *ACM/IEEE International Symposium on Computer Architecture*, p. 266–277, 2001.

[7] T. Nowatzki, J. Menon, C. Ho, and K. Sankaralingam, "Architectural simulators considered harmful," *IEEE Micro*, vol. 35, no. 6, pp. 4–12, 2015.

[8] J. D. McCalpin, "HPL and DGEMM performance variability on the Xeon platinum 8160 processor," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 225–237, 2018.

[9] L. Wang, Y. Chang, and K. Cheng, *Electronic Design Automation: Synthesis, Verification, and Test*. Morgan Kaufmann Publishers, 2009.

[10] C.-K. Cheng, J. Lillis, S. Lin, and N. Chang, *Interconnect analysis and synthesis*. New York, NY: Wiley Interscience, 2000.

[11] L. T. Pillage and R. A. Rohrer, "Asymptotic waveform evaluation for timing analysis," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, pp. 352–366, Apr. 1990.

[12] W. C. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers," in *Journal of Applied Physics*, vol. 19, pp. 55–63, Jan. 1948.

[13] H. Chang and S. S. Sapatnekar, "Statistical timing analysis considering spatial correlations using a single PERT-like traversal," in *IEEE International Conference On Computer Aided Design*, pp. 621–625, 2003.

[14] C. J. Alpert, J. Hu, S. S. Sapatnekar, and C.-N. Sze, "Accurate estimation of global buffer delay within a floorplan," in *IEEE International Conference On Computer Aided Design*, pp. 706–711, 2004.

[15] A. B. Kahng, S. Kang, H. Lee, S. Nath, and J. Wadhwani, "Learning-based approximation of interconnect delay and slew in signoff timing tools," in *ACM/IEEE International Workshop on System Level Interconnect Prediction*, vol. 00, pp. 1–8, 2014.

[16] S. S. Han, A. B. Kahng, S. Nath, and A. S. Vydyanathan, "A deep learning methodology to proliferate golden signoff timing," in *Design, Automation & Test in Europe*, pp. 260:1–260:6, 2014.

[17] A. B. Kahng, M. Luo, and S. Nath, "Si for free: machine learning of interconnect coupling delay and transition effects," in *ACM/IEEE International Workshop on System Level Interconnect Prediction*, pp. 1–8, 2015.

[18] W. J. Chan, K. Y. Chung, A. B. Kahng, N. D. MacDonald, and S. Nath, "Learning-based prediction of embedded memory timing failures during initial floorplan design," in *Asia and South Pacific Design Automation Conference*, pp. 178–185, 2016.

[19] Y.-C. Lu, J. Lee, A. Agnesina, K. Samadi, and S. K. Lim, "Gan-cts: A generative adversarial framework for clock tree prediction and optimization," in *IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–8, 2019.

[20] S. Davidson, "Characteristics of the ITC99 benchmark circuits," in *IEEE International Test Synthesis Workshop*, 1999.

[21] J. Knudsen, "Nangate 45nm open cell library," in *CDNLive, EMEA*, 2008.

[22] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. CreateSpace, 2009.

[23] F. Santosa and W. Symes, "Linear inversion of band-limited reflection seismograms," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 4, pp. 1307–1330, 1986.

[24] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," in *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[26] F. Chollet *et al.*, "Keras." `https://keras.io`, 2015.

[27] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359 – 366, 1989.

[28] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill, "Architecture validation for processors," in *ACM/IEEE International Symposium on Computer Architecture*, pp. 404–413, 1995.

[29] Intel Corporation, "11th generation Intel® Core™ i7 processors." https://ark.intel.com/content/www/us/en/ark/products/series/202986/11th-generation-intel-core-i7-processors.html.

[30] Advanced Micro Devices Inc., "AMD Ryzen™ 9 5950X." https://www.amd.com/en/products/cpu/amd-ryzen-9-5950x#product-specs.

[31] ARM Ltd., "Cortex-A710." https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a710.

[32] "SPEC CPU2006." https://www.spec.org/cpu2006.

[33] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[34] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation Intel Core: New microarchitecture code-named Skylake," *IEEE Micro*, vol. 37, no. 2, pp. 52–62, 2017.

[35] P. Gepner, D. L. Fraser, and V. Gamayunov, "Evaluation of the 3rd generation Intel Core processor focusing on HPC applications," in *International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 1–6, 2012.

[36] D. B. Noonburg and J. P. Shen, "Theoretical modeling of superscalar processor performance," in *IEEE/ACM International Symposium on Microarchitecture*, pp. 52–62, 1994.

[37] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *ACM/IEEE International Symposium on Computer Architecture*, pp. 338–349, 2004.

[38] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Transactions on Computer Systems*, vol. 27, no. 2, pp. 1–37, 2009.

[39] M. J. Clement and M. J. Quinn, "Analytical performance prediction on multicomputers," in *ACM/IEEE Conference on Supercomputing*, pp. 886–894, 1993.

[40] B. Black and J. P. Shen, "Calibration of microprocessor performance models," *Computer*, vol. 31, no. 5, pp. 59–65, 1998.

[41] E. Ould-Ahmed-Vall, J. Woodlee, C. Yount, K. A. Doshi, and S. Abraham, "Using model trees for computer architecture performance analysis of software applications," in *IEEE International Symposium on Performance Analysis of Systems & Software*, pp. 116–125, 2007.

[42] R. Joseph and M. Martonosi, "Run-time power estimation in high performance microprocessors," in *ACM/IEEE International Symposium on Low power electronics and design*, pp. 135–140, 2001.

[43] G. Contreras and M. Martonosi, "Power prediction for Intel XScale processors using performance monitoring unit events," in *ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 221–226, 2005.

[44] W. L. Bircher and L. K. John, "Complete system power estimation: A trickle-down approach based on performance events," in *IEEE International Symposium on Performance Analysis of Systems Software*, pp. 158–168, 2007.

[45] S. Yang, Z. Luan, B. Li, G. Zhang, T. Huang, and D. Qian, "Performance events based full system estimation on application power consumption," in *IEEE International Conference on High Performance Computing and Communications*, pp. 749–756, 2016.

[46] P. Bose, "Architectural timing verification and test for super scalar processors," in *IEEE International Symposium on Fault-Tolerant Computing*, pp. 256–265, 1994.

[47] S. Surya, P. Bose, and J. Abraham, "Architectural performance verification: PowerPC processors," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 344–347, 1994.

[48] K. Shen, M. Zhong, and C. Li, "I/O system performance debugging using model-driven anomaly characterization.," in *USENIX Conference on File and Storage Technologies*, vol. 5, pp. 309–322, 2005.

[49] B. Cook, T. Kurth, B. Austin, S. Williams, and J. Deslippe, "Performance variability on Xeon Phi," in *IEEE International Conference on High Performance Computing*, pp. 419–429, 2017.

[50] D. Skinner and W. Kramer, "Understanding the causes of performance variability in HPC workloads," in *IEEE Workload Characterization Symposium*, pp. 137–149, 2005.

[51] N. Utamaphethai, R. S. Blanton, and J. P. Shen, "A buffer-oriented methodology for microarchitecture validation," *Journal of Electronic Testing*, vol. 16, no. 1-2, pp. 49–65, 2000.

[52] A. Adir, H. Azatchi, E. Bin, O. Peled, and K. Shoikhet, "A generic micro-architectural test plan approach for microprocessor verification," in *ACM/IEEE Design Automation Conference*, pp. 769–774, 2005.

[53] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Computing Surveys*, vol. 48, no. 1, pp. 1–35, 2015.

[54] C. Killian, K. Nagaraj, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala, "Finding latent performance bugs in systems implementations," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 17–26, 2010.

[55] D. J. Dean, H. Nguyen, and X. Gu, "UBL: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems," in *IEEE International Conference on Autonomic Computing*, pp. 191–200, 2012.

[56] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 19–33, 2019.

[57] A. Nistor, T. Jiang, and L. Tan, "Discovering, reporting, and fixing performance bugs," in *IEEE/ACM Working Conference on Mining Software Repositories*, pp. 237–246, 2013.

[58] M. Alam, J. Gottschlich, N. Tatbul, J. S. Turek, T. Mattson, and A. Muzahid, "A zero-positive learning approach for diagnosing software performance regressions," in *Advances in Neural Information Processing Systems*, pp. 11623–11635, 2019.

[59] J. Li, Y. Chen, H. Liu, S. Lu, Y. Zhang, H. S. Gunawi, X. Gu, X. Lu, and D. Li, "Pcatch: Automatically detecting performance cascading bugs in cloud systems," in *ACM European Conference on Computer Systems*, pp. 1–14, 2018.

[60] R. Atachiants, G. Doherty, and D. Gregg, "Parallel performance problems on shared-memory multicore systems: taxonomy and observation," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 764–785, 2016.

[61] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *International Conference on Software Engineering*, pp. 1013–1024, 2014.

[62] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: A comparative study of generic and composite text models," in *IEEE/ACM Working Conference on Mining Software Repositories*, p. 43–52, 2011.

[63] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *International Conference on Software Engineering*, pp. 14–24, 2012.

[64] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *IEEE/ACM International Conference on Automated Software Engineering*, pp. 345–355, 2013.

[65] S. Akbar and A. Kak, "Scor: Source code retrieval with semantics and order," in *IEEE/ACM International Conference on Mining Software Repositories*, pp. 1–12, 2019.

[66] C. Delimitrou and C. Kozyrakis, "iBench: Quantifying interference for datacenter applications," in *IEEE International Symposium on Workload Characterization*, pp. 23–33, 2013.

[67] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *ACM European Conference on Computer Systems*, pp. 1–14, 2014.

[68] C. Delimitrou, D. Sanchez, and C. Kozyrakis, "Tarcil: Reconciling scheduling speed and quality in large shared clusters," in *ACM Symposium on Cloud Computing*, pp. 97–110, 2015.

[69] "SPEC CPU2017." `https://www.spec.org/cpu2017`.

[70] T. M. Conte, M. A. Hirsch, and K. N. Menezes, "Reducing state loss for effective trace sampling of superscalar processors," in *IEEE International Conference on Computer Design. VLSI in Computers and Processors*, pp. 468–477, 1996.

[71] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *ACM/IEEE International Symposium on Computer Architecture*, pp. 84–97, 2003.

[72] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 318–319, 2003.

[73] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *ACM SIGPLAN Notices*, vol. 37, no. 10, pp. 45–57, 2002.

[74] G. Hamerly, E. Perelman, and B. Calder, "How to use simpoint to pick simulation points," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 4, pp. 25–30, 2004.

[75] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[76] Intel Corporation, "Intel Xeon processor scalable family: Specification update." `https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-scalable-spec-update.pdf`, 2019.

[77] NXP Semicondutctors, "Chip errata for the MPC7448." https://www.nxp.com/docs/en/errata/MPC7448CE.pdf, 2008.

[78] Texas Instruments, "AM3517, AM3505 Sitara processors silicon revisions 1.1, 1.0: Silicon errata." http://www.ti.com/lit/er/sprz306e/sprz306e.pdf, 2016.

[79] Intel Corporation, "Intel386™ DX processor: Specification update." http://www.nj7p.org/Manuals/PDFs/Intel/272874-003.pdf, 2004.

[80] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time series," in *The handbook of brain theory and neural networks*, 1995.

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

[82] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference for Learning Representations*, 2015.

[83] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794, 2016.

[84] T. Hastie and R. Tibshirani, "Classification by pairwise coupling," *The annals of statistics*, vol. 26, no. 2, pp. 451–471, 1998.

[85] E. L. Allwein, R. E. Schapire, and Y. Singer, "Reducing multiclass to binary: A unifying approach for margin classifiers," *Journal of machine learning research*, vol. 1, no. Dec, pp. 113–141, 2000.

[86] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[87] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," in *Annals of Statistics*, pp. 1189–1232, 2001.

[88] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Po-

lat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," in *Nature Methods*, vol. 17, pp. 261–272, 2020.

[89] C. E. Shannon, "Communication in the presence of noise," *Proceedings of the Institute of Radio Engineers*, vol. 37, no. 1, pp. 10–21, 1949.