AUTOMATED DETECTION OF MEMORY PERFORMANCE BUGS IN MICROPROCESSORS

A Thesis

by

SARA MARIAM JACOB

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

MASTER OF SCIENCE

Chair of Committee,	Jiang Hu
Co-Chair of Committee,	Paul Gratz
Committee Member,	Eun Jung Kim
Head of Department,	Miroslav M. Begovic

August 2020

Major Subject: Computer Engineering

Copyright 2020 Sara Mariam Jacob

ABSTRACT

An underutilization of technological advancements and loss of investment is experienced when performance degradation happens. Processors constitute most modern devices, and a detriment in processor performance would consequently impact the pace of our everyday life. Ensuring the highest possible processor performance becomes increasingly significant. Performance bugs are design anomalies that degrade performance of a processor and differ from functional logic bugs in that they do not affect functionality or program result. Efforts to detect performance bugs are often time-intensive and involves manual analysis. With the two phase methodology proposed in this thesis, automated detection of performance bugs is possible. The first phase comprises of a performance predictor using machine learning while the second phase uses binary classification in a bug detection scheme.

Lack of a reference standard denoting correct performance for a new processor design poses among the challenges in performance bug detection. The first phase of the methodology resolves this by predicting expected performance for a new microarchitecture release after training on past microprocessor generations. Deviance of predicted performance from actual processor performance signals performance regressions. Errors based on this deviance are passed to the second phase of the technique that employs a binary classifier in a voting based approach, labeling the architecture as with or without performance bug. The focus of this work is on performance bugs in the memory system of microprocessors. As markers of processor performance, two assist metrics are used, Instructions Per Cycle (IPC) and Average Memory Access Time (AMAT).

The method provides promising results - using LSTM for machine learning in the predictor, 100% overall accuracy is achieved in bug detection with IPC and 83% with AMAT, while XGBoost predictor has an accuracy of 100% in classification for both IPC and AMAT.

DEDICATION

To Pappa and Mamma.

ACKNOWLEDGMENTS

The graduate school experience has been enriching, and this journey of completing my masters has been one of perseverance and arduous effort. The two years at Texas A&M University has offered me various opportunities to grow and nurture myself, helping me imbibe certain personality traits that I appreciate and hope to cherish in my future endeavors as well. I thank my research advisor, Dr.Jiang Hu, for his motivation, mentoring and rendering support through both the good and difficult times through the course of my research. His words of encouragement, patience and tenacity to strive ahead helped me aspire further and challenge myself. I would like to express my gratitude to my co-chair, Dr.Paul Gratz, for his valuable insights and academic guidance through the duration of my thesis work. I extend appreciation for his affable demeanor and considerate nature that helps instill confidence. I would also like to thank Dr.Eun Jung Kim who is on my committee for her time and feedback.

I would like to thank my research teammates Erick, Tony and Kyungrak for their support. Erick has provided me with timely advice through the project and helped me learn a lot from his research expertise. I would also like to thank Mahesh Ketkar and Michael Kishinevsky from Intel for their valuable feedback. I thank my friends for their cheer and care. I thank my parents and family for their unconditional support and unwavering faith in me.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Jiang Hu and Professor Paul Gratz of the Department of Electrical and Computer Engineering and Professor Eun Jung Kim of the Department of Computer Science and Engineering.

This work derives on the ideation from my research team working on automated performance bug detection for various areas of the processor and Erick C. Barboza's methodology related to processor core.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

This thesis work was not in specific funded by any source and there are no direct contributors to acknowledge. The student received a Graduate Teaching Assistantship from the Department of Electrical and Computer Engineering. The research project from the team as a whole is supported by the Semiconductor Research Corporation (SRC).

NOMENCLATURE

IPC	Instructions Per Cycle
AMAT	Average Memory Access Time
ML	Machine Learning
LSTM	Long Short Term Memory
ROC	Receiver Operating Characteristics
MSE	Mean Squared Error
TSE	Timestep Error
SPP	Signature Path Prefetcher
HPC	High Performance Computing
CNN	Convolutional Neural Networks
LRU	Least Recently Used
RFO	Read For Ownership

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

1.1 Background and Motivation

Since the advent of the microprocessor [1], there has been a constant drive to improve processor performance. Over the decades, the three main technology factors that have boosted microprocessor performance are: transistor scaling, core micro-architecture techniques and cache memories [2]. Each new generation of microprocessor released in industry implements certain micro-architectural advances in order to boost performance, typically to increase processor execution speed and operate at higher clock rates. Processor execution speed which represents cumulative processor performance is influenced not just by the simple execution speed of functional units, but depends on various other micro-architectural design features that contribute to total execution time. There have been many micro-architectural techniques aimed at improving performance by decreasing total time that an instruction takes to commit from the stage it is fetched. The introduction of cache hierarchies to combat Memory Wall [3] [4], practice of instruction caching (trace cache), speculation in out-of-order processors are all instances aimed toward the same goal. Designs such as Intel's Pentium4 [5] were highlighted on its ability to run at faster clock rates than its predecessors. Every new generation of processors or latest tape-outs aim to perform faster than their predecessors and implements certain micro-architectural changes or advancements compared to previous designs to achieve this end.

There have been many efforts in both the industry and academia for years now to ensure logical correctness of results produced by processors. A processor incapable of producing correct results for a program or block of code tends to be useless in real life. As a consequence, ventures in pre-silicon verification and post-silicon validation to eliminate logic functional bugs [6] [7] have garnered interest for long. However, the processor execution speed can be affected by anomalies in the micro-architecture that manifest as 'performance bugs'. Ventures in this domain have been few and works aimed at detecting performing bugs using set methodologies are fairly recent.

On a single core out-of-order processor, the instructions executed per cycle (IPC) is a fairly good marker of processor performance. By understanding the dependence of IPC on the micro-architectural implementation across various generation of processors, we propose that on seeing a new architecture configuration it is possible to predict the expected IPC for that micro-architecture design. Average Memory Access Time (AMAT) is another indicator or processor performance and provides insight into the memory system in specific. The reasoning used for IPC is extended for AMAT as well in this work.

While designing modern-day processors with such a vast spectrum of micro-architectural aspects and multiple features involved, performance bugs may creep in. 'Performance Bugs' are bugs in the micro-architecture that detriment the processor performance. By bugs, we mean any and all anomalies in the design implementation that cause the processor to behave differently from how the micro-architect originally intended it to. The task of detecting performance bugs is challenging due to the lack of a comparable standard that denotes correct performance for a design. Unlike logic bugs which manifest by producing incorrect values for a code block (among other ways), performance bugs mainly impact execution speed and is indicated in our work by IPC or AMAT. IPC and AMAT serve as assist metrics in our work. While performance bugs impact the assist metric and decrease performance from a bug-free version of the same architecture, this impact would be significantly lesser than the overall increase in the assist metric as compared to a previous generation micro-architecture. Another challenge is that performance degradation due a bug might appear only on select workloads, this is due to the fact that behavior of different applications would be different. All applications need not necessarily be impacted by the same performance bug. These two factors, not knowing the achievable performance for a new processor design and varying workload behavior, make detection of performance bugs very difficult.

1.2 Objective

The objective of this thesis is to propose a method to detect performance bugs present in memory systems of microprocessors and automate this process. We introduce a two-phase methodology to detect performance bugs. While the method may be applicable to all components of a processor,

our focus is on processor performance bugs present in the uncore components, specifically cache and memory. The bugs used to test the method, the selection of simulator have all been done keeping in mind our specific focus on memory performance bugs.

Following are the tasks undertaken to achieve our objective:

- Create bug cases specific to memory systems on microprocessors that affect performance in varying degrees. We can segment these performance bugs as those with high, medium, low and very low impact depending on the performance penalty they cause.

- Devise a set of architectures to run the benchmark suite on. For each architecture, micro-architectural specifications are configured on the simulator to relate closely with the respective processor architecture in industry.

- Develop performance counters on the simulator that extract and record information critical to us on each simulation run. Information recorded by these counters is selectively used as features and fed to the machine learning model.

- Develop a machine learning model capable of learning efficiently from a set of architectures for a single application, and then from the training attained be able to predict performance (IPC or AMAT) on a new set of architectures.

- Develop a binary rule-based classifier capable of classifying the design as having or not having a bug using inputs obtained from the first phase of the machine learning methodology.

1.3 Document Organization

The remainder of this document is organized as follows. Section 2 details previous and related works that could be associated in terms of methodology aspects with this work. Section 3 provides an overview of the methodology used and then delves into further details and attributes along with other paraphernalia. Section 4 outlines the Phase1 of the proposed methodology with details on the machine learning engines used. Section 5 focuses on Phase2, detailing the binary classifier for bug detection and mentions the classification conditions used. Section 6 shows experimental

results of the methodology and overall bug detection capability. Section 7 concluded the thesis and talks about future scope.

2. PREVIOUS WORK

Performance debugging in various arenas has been a topic of interest in research, owing to the significant improvement in overall throughput or productivity provided with successful debug efforts. Whilst ventures specific to hardware aimed at employing automated techniques for performance debugging from a micro-architectural perspective are limited, related works and techniques suggested in other domains such as cloud computing, software etc. provide instrumental insights.

Many performance bugs appear as an aftermath of unexpected temporal interleaving of events [8]. The paper [8] views the task of detecting performance bugs to be composed of the steps of determining expected execution of a system, finding the anomalous executions and then inspecting the anomalous executions to localize bug. An instance of real-case performance bug discovered on the Xeon Platinum 8160 processor has been outlined by McCalpin [9]. Initially observed as a performance drop on few nodes of the cluster while testing for HPC workloads, this performance variability was judged conclusively to arise from a hardware anomaly after multiple runs on the cluster using various benchmarks. Efforts to determine bug presence were time intensive and strenuous. The consecutive task of localizing the bug and determining its root cause involved indepth analysis posing a further challenge. The root cause of performance degradation was traced to excessive snoop filter evictions due to associativity conflicts; the analysis was aided by performance monitoring infrastructure constituent of performance counters. Albeit successfully finding the root cause of a daunting performance bug, the laborious efforts involved highlight the need for a streamlined, direct and automated approach to detect performance bugs in hardware.

In a recent paper [10], an automated performance debugging system is deployed to signal quality of service violations in cloud microservices. The authors prescribe data driven performance diagnostics as the path toward understanding system behavior and performance; as can be seen with this proposition many aspects in the paper make sense in light of our goal as well. In the paper, two levels of tracing is carried out to collect data that would aid in comprehending system performance - performance counters are used in a lower level tracing while tunable microbench-

marks were used to grasp resource contention in higher level tracing. Deep learning employing a combination of convolutional neural networks (CNN) and LSTM is used to learn spatial and temporal patterns on past quality of service violations and the system is trained. Machine learning is counted upon as a reliable way to learn from past instances and help a system operate using that learning for the future. The learning gathered enables the debug engine to signal upcoming quality of service violations during runtime and allows cluster manager to take measures that prevent the performance degradation from happening.

In software, functions common and unchanged between successive versions of software are deduced to not be the cause of performance regressions; modified functions blocks in a new software version relative to a former version are analyzed for being possible sources of performance degradation on the new program code. The system is trained on negative or non-anomalous samples and made to learn this data distribution in the zero positive learning approach, test samples that deviate sufficiently from these are judged to be positive or anomalous samples [11]. Profiling is carried out with hardware telemetry using performance counters, and the performance signatures collected from the functions of interest in the later and former software versions are used for training and test respectively. Deviation of a test sample from the non-anomalous sample distribution is checked via reconstruction error. Certain parallels could be drawn with our work from this similar attempt to automate detection of performance regressions in software. Another effort at detecting performance bugs in software proposes a rule-based detection strategy [12] based on their characteristics study of several real-case performance bugs in software suites.

The lack of a believable reference model or standard to check performance against has forever served as a challenge in pre-silicon verification as noted by Bose et. al in their work [13]. Early attempts at performance debug suggested an integrated validation approach, combining functional and performance verification. Case was made for augmenting the functional test suite with simulation based test cases for performance verification [13], using single instruction test cases that checked per stage pipeline latencies. The timing information obtained is checked against the expected performance indicated through cycle-based timing simulators or other methods of estimat-

ing timing of the processor. To obtain accurate timing estimation, prior efforts were made toward validating architectural timing models used to predict processor performance [14].

Most of these existing techniques aim at combating the issue of performance degradation in different domains, and the few that inspect performance bugs in the processor realm are deficient of a direct and automated approach that is capable of easily detecting performance bugs without significant manual intervention. As seen through this section, while many of the cited works involve revolutionary techniques directed toward making progress in the arena of performance debugging, none of them reliably address the issue of detecting the presence of performance anomalies in microprocessors owing to design factors. This need is sought to be resolved by our method.

3. METHODOLOGY

3.1 Overview

The goal of this work is to detect performance bugs in memory systems of microprocessors, and to automate this process. We introduce a two-phase machine learning methodology depicted in Figure 3.1 that aims to achieve the same.

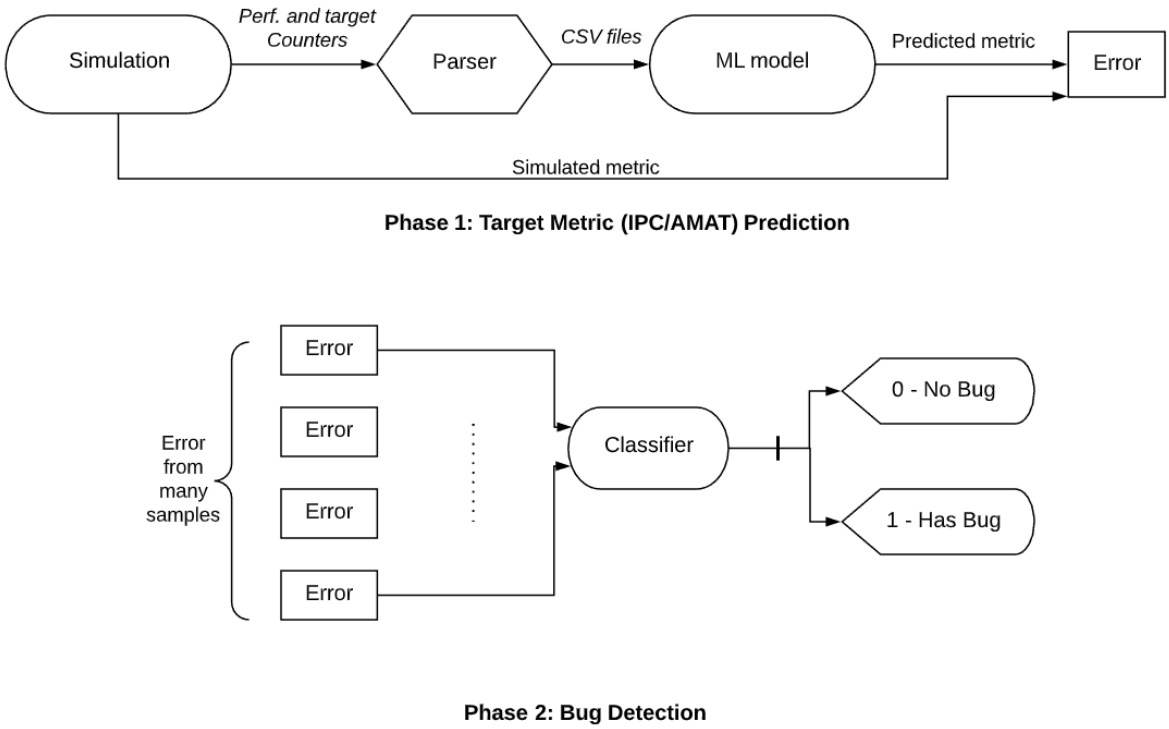

Figure 3.1: Overview of Two-phase Methodology

In the first phase of the method, we use the micro-architecture simulator ChampSim [15] to simulate the microprocessor behavior. The configured micro-architectures are run against traces taken from the SPEC [16] benchmark suite on ChampSim. The output at the end of each simulation run contains information from the performance counters and the IPC/AMAT counter recorded after

warmup at intervals of 500,000 cycles. The simulation output also includes cumulative statistics such as total accesses for each cache level, cumulative IPC and speculative prediction accuracy. This information obtained via counters is extracted using a parser and converted into files readable by the machine learning algorithm. The files created are also padded to attain same sequence length for all architectures under a single trace. Pre-padding of the sequences are done to create samples of equal length. We run simulations of architectures both with and without bugs. For the purpose of testing and providing evidence for our method, we insert performance bugs formulated by us with reference to real-case bug scenarios [6] [7] in few architectures. The performance bugs have been formulated to affect cache and memory operations [17] in the micro-processor and cause a resultant performance penalty indicated by a drop in IPC/AMAT. Though we know which architectures have the bug for the purpose of our work, this information has not been used in our methodology. As far as the technique we propose is concerned, the simulator, LSTM model and classifier make no distinction while treating bug-free architectures or those with bug; our method would behave in similar respects if it were provided architectures for bug detection with performance bugs of any sort.

The first phase of the methodology uses a ML model to predict the assist metric (IPC or AMAT) for architectures to be tested. The ML engines used in our case are LSTM (long short term memory) [18] and XGBoost [19] as they are suited for our work and dataset, and validate that choice with promising results. The performance counters are fed as features to the ML model and IPC/AMAT is the target metric for the model. The counters which have maximum influence over the target metric are selected and only those counters are used as features to train and test. Once training and validation is completed, testing is done on the sample set with architectures to test. The IPC/AMAT for these architectures is predicted and the variance or error of this predicted metric from the actual metric obtained from simulation is calculated. This difference between actual and predicted IPC/AMAT for each architecture forms the basis of our premise to detect the bug. In theory, the larger the error, the greater the possibility of a bug.

To confirm the presence of the bug and to automate it, we apply phase two of our method-

ology. A binary classifier is used to classify the architectures as bug-free or one that contains a performance bug. The binary classifier is implemented as a rule-based classifier and the error metric obtained from various architecture samples used under test in phase1 are fed as inputs. Using these inputs the classifier puts each architecture into one of two categories; it outputs a 0 for architecture which does not have bug and outputs 1 for architecture which it detects to possess a performance bug.

3.2 Bug Development

The aim was to create bug cases that cause varying impact on the memory systems in microprocessors and thus affect performance. Most of the bugs have been developed with reference to real-case scenarios and industry examples [6] [7] [20]. Bugs 1 and 2 in Table 3.1 pertain to replacement of cache blocks in memory and introduce anomalous behavior that degrades performance. Bugs 3, 4 and 8 are each bug effects coupled with certain bug activation criteria. Bug activation criterion refers to a set of conditions that trigger the bug effect into action; this is analogous to real-life performance bugs that often come into play when activated by some other triggering factor or event in the processor [6]. Bugs 5, 6 and 7 introduce anomalous behavior in the prefetcher and diminish processor performance.

The age counter tracks the oldest block in the cache and helps select this oldest block as the victim for eviction during replacement. The age counter of each block increments its value for every cache access that does not access that block. The block with the most recent access has its age counter reset to 0. A departure from this behavior is created by Bug1 where it prevents the age counter from being reset on an access. The replacement policy used in our architecture configurations is the Least Recently Used (LRU) replacement policy. Bug2 selects the youngest block as the victim for replacement. This means the most recently used cache line would be evicted, while the probability that this line would be needed again in the future is high (principle of temporal locality) leading to unnecessary movement of blocks between cache and main memory which takes a toll on performance.

Bug3, 4 and 8 from Table3.1 couple an activation criteria with their respective bug effects.

Area	Bugs	Description
Replacement	Bug1	Age counter does not reset after access
Replacement	Bug2	Most recently used block becomes victim for replacement
Cache Operations	Bug3 (4 variants)	After the first 2 load misses in L1D, every X clock cycles the read operation is delayed by Y clock cycles.
Cache Operations	Bug4 (2 variants)	After first two writeback misses, read operation to cache delayed by Y cycles at intervals of X clock cycles. Implemented for second level cache.
Cache Operations	Bug8	Every 200 clock cycles for more than 2 cache misses, subsequent read to cache delayed by 50 cycles.
Prefetcher	Bug5	Prefetcher forgets page offset for delta. Ends up at wrong address
Prefetcher	Bug6 (2 variants)	Lookahead path takes least confident entry. Does not account for previous path confidences 50% of time.
Prefetcher	Bug7	Prefetches dropped and not added to filter, are counted as been prefetched

Table 3.1: Bug Descriptions

Bug3 is activated after the first two load misses to the L1D cache is seen. From the third load miss onwards, every X clock cycles the subsequent read operation to L1D is delayed by Y clock cycles. X and Y are parameters that are varied across the bug variants of Bug3 to create bug versions with varying impact. Class of Bug3 has 4 variants, each activated by seeing a certain type of miss and having different values for X and Y. The first two variants are activated after seeing the first two load misses to the L1D cache. Variant1 delays the read operation seen every 10 clock cycles by 10 cycles while variant2 delays the read access seen every 20 cycles. Variants 3 and 4 of Bug3 are activated after the first two Read For Ownership (RFO) misses to the L1D cache. Value of X is 20 and 10 in variants 3 and 4 respectively, while the read operation is delayed by 10 cycles in both cases. Bug4 is similar to Bug3 but implemented for the L2C. There are two variants of Bug4, each activated after the first two writeback misses to the second level cache. The first variant takes the read operation to the L2C seen every 30 clock cycles and delays it by 30 cycles. The second variant of Bug4 delays both read and writeback operations by 10 clock cycles, and this is done at an interval of 10 cycles. Bug8 is activated only if more than 2 cache misses are seen in total from

all the access types (load, RFO, writeback and prefetch). This is checked every 200 clock cycles and if the condition is satisfied by the L1D cache during that interval, then the subsequent read to L1D is delayed by 50 cycles.

Bugs 5,6 and 7 introduce anomalous behavior in the prefetcher that affects performance. Our architecture configurations use the Signature Path Prefetcher (SPP) where signatures created from deltas form the basis of the prefetch mechanism [21]. The deltas obtained from the access requests and by prediction for lookahead prefetches (speculative prefetches using predicted deltas beyond the demand request) are used to create signatures that store this accumulated history of access patterns and are finally used to prefetch. Bug5 causes the page offset to not be used in calculation of delta. This makes the prefetcher essentially use the wrong address to prefetch.

For Bug6 we deviate performance from the normal prefetcher by affecting the confidence values. In lookahead prefetching, the confidence in the prefetch requests must decrease as the product of confidences along the speculative prefetch path [21]. Among the possible speculative paths, the prefetch is carried out along the most confident entry in the bug-free implementation. Bug6 causes the prefetcher to select the least confident path for lookahead prefetch and 50% of the time does not take into account the previous path confidences. This would mean a prefetch that should have been deemed as less accurate and marked with lesser confidence value is now marked with the local confidence value without accounting for the path confidences that led to that particular prefetch. There are two variants to Bug6 that differ in the local confidences assigned 50% of the time. Bug7 is a realistic bug that came across after the Data Prefetching Championship for which the Signature Path Prefetcher (SPP) method [21] was an entry and ranked among the top prefetchers. Bug7 drops certain prefetches but incorrectly marks them as being prefetched. This bug impacts performance since dropped prefetches are not added to the filter correctly but considered as having been prefetched.

The performance impact on a per bug basis for both the IPC and AMAT metrics have been shown in Tables 3.2 and 3.3. Variants of a specific bug type are ordered sequentially; for instance we have Bug4a, Bug4b denoting the first and second variants of Bug4 respectively in Tables 3.2

Bugs	IPC Impacts%	AMAT Impacts%
Bug1	7	17
Bug2	7	36
Bug3a	7	4
Bug3b	3	2
Bug3c	3	2
Bug3d	6	3
Bug4a	9	28
Bug4b	2	7
Bug8	0.8	0.7

Table 3.2: Bug Impacts Without Prefetch
The bug impacts have been rounded to nearest digit and displayed

Bugs	IPC Impacts%	AMAT Impacts%
Bug1	4	28
Bug2	8	60
Bug3a	8	4
Bug3b	3	1
Bug3c	3	1
Bug3d	8	3
Bug4a	12	53
Bug4b	2	11
Bug5	4	15
Bug6a	2	7
Bug6b	0.6	2
Bug7	0.1	0.3
Bug8	1	0.5

Table 3.3: Bug Impacts With Prefetch
The bug impacts have been rounded to nearest digit and displayed

and 3.3. Speedup is measured of bug over no-bug for a trace and then geometric mean is taken across all the traces to calculate impact of a bug. For instances where there is no speedup (the particular trace does not see a performance penalty) or there is a slight performance hike (when a particular workload might perform better with bug), geometric mean is taken following the norm as when dealing with zeroes or negative values.

We have categorized the bugs into 4 sections based on their impacts. The four impact bins are

Very Low, Low, Medium and *High*.*Very Low* impact means an impact of less than 2% on IPC or AMAT. A *Low* impact refers to when the performance degradation is between 2-5%. From 5-9% it is taken as *Medium* impact and bugs resulting in above 9% performance degradation is categorized as *High* impact. Figures 3.2 and 3.3 depict the distribution of bugs by impact.

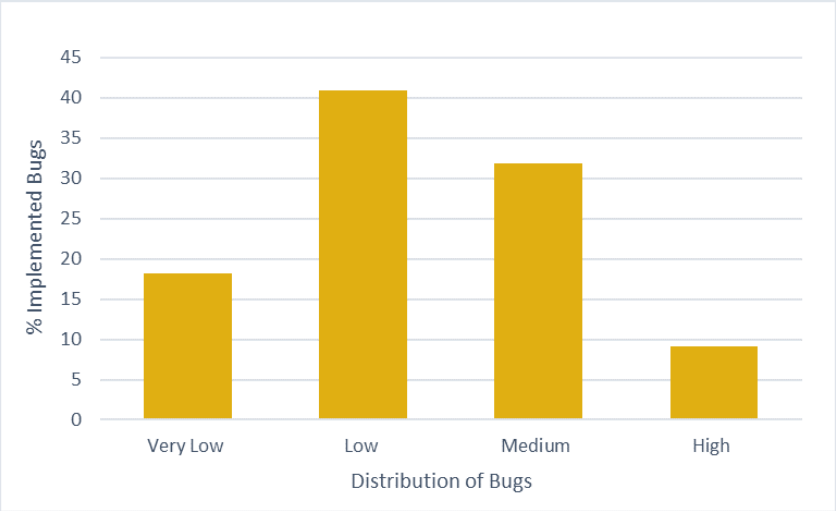

Figure 3.2: Distribution of IPC Bugs in Impact Bins

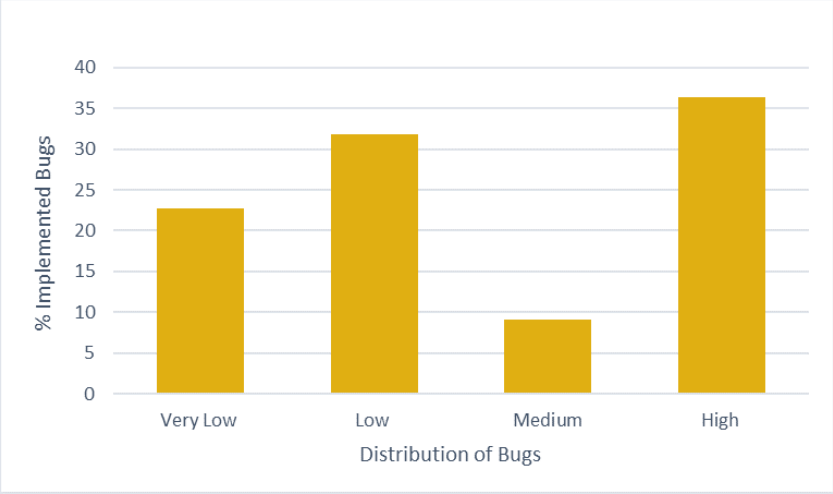

Figure 3.3: Distribution of AMAT Bugs in Impact Bins

It can be observed that the high category impact bin is more populated in AMAT as compared to IPC. This is since many bugs in the medium low impact bins for IPC are kicked to the high impact bin in AMAT. This is expected since, apart from the fact that same bugs would have different impact on each metric, the impact on AMAT would be amplified as compared to IPC in general. For most memory bugs, AMAT would be affected more since the metric the directly associated with memory operations while IPC is constituent of AMAT and many other influencing factors. One could also observe that Very Low bin in AMAT also grows in population. This is due to a few bugs transferring from the increased impact bins in IPC and displaying reduced impact for AMAT. This behavior is exhibited by Bug3 and its variants; Tables 3.2 and 3.3 note this as well where unlike other bugs which have greater impact on AMAT compared to the corresponding IPC effect, Bug3 tones down the impact on AMAT relative to IPC. Bug3 stalls the read operation in the L1D cache upon activation. While this would impact AMAT, it also delays the entire execution pipeline for an instruction and delays the completion of instruction. Overall this bug would tend to produce a larger impact on IPC; while AMAT does influence IPC, the effect on other constituent components of IPC are higher in this case.

3.3 Architecture Configurations

To train and test out predictor we configure the micro-architecture simulator *ChampSim* to a mix of both real, known architectures and artificial architectures. Table 3.4 displays the 12 base architecture configurations implemented for the purpose of this work. Each configuration has been implemented with and without prefetching capability, to provide a set of 24 architectures in total. The metric column in Table 3.4 details the various specifications such as clock period of the processor, number of re-order buffer entries, cache settings for each level, branch mis-prediction penalty etc. that are varied to achieve each configuration. The 8 real and known architectures used are Broadwell, Haswell, Skylake, Sandybridge, Ivybridge, AMD K10, AMD Ryzen7 and Nehalem. These are configured to relate closely with observed industry specifications [22] and replicate the corresponding real processor architectures. The 4 artificial architectures denoted by R0, R1, R2 and R3 in Table 3.4 have been configured by varying the metrics by keeping realistic settings in

mind. These architectures form the set for training the predictor models as well as testing the predictor with samples that were unseen during the training phase. The effort has been to include a range of architectures spanning from older to more recent processor generations as a parallel to real-life performance debug situation relying on our technique. The training of the predictor would be carried out on the predecessor architectures while testing for possible performance bugs would be performed on the new processor generation.

The bugs for testing have been implemented on Skylake architecture. Haswell (without prefetch) and Sandybridge (with prefetch) architectures without bug implementations have also been used for testing. The remaining set of 20 architectures are used for training and validation.

3.4 Performance Counters

Prior ventures in the arena of performance debugging for various purposes [10] [9] have used performance counters as a reliable source to understand system behavior, utilize for performance estimation and aid in debug. Data-driven performance analysis, made possible with performance counters have been shown to help a system identify performance anomalies better [10]. Performance counters are essentially values of metrics collected at regular intervals, that each accentuates certain aspects of system behavior and provide insight into that behavior over a span of time. Both IPC and AMAT are recorded in counters and used as targets to the predictor models. In our AMAT calculation, the L1 latencies are not taken into account owing to the operations of the simulator. However, since this latency in the L1D is ignored irrespective of hit or miss, the behavior of AMAT remains same and is not reckoned to vary any insights gained from the results.

While many performance counters are viable to understand processor performance we focus on counters that detail memory system behavior. The counters needed for our work were built into the simulator since the original version of *ChampSim* did not contain sufficient counters to aid our study. Our counters are based on the total number of accesses, hit ratios and timing information on a cache for each type of access. An access to a cache could be a load request, Read For Ownership (RFO), writeback or prefetch request. We have a total of 64 counters that help detail cache behavior and access patterns for a certain benchmark and architecture configuration. The counters

Metric	BW[a]	HW	SL	SB	IB	K10	Ryz7	NH	R0	R1	R2	R3
Clock	4GHz	3.4GHz	4GHz	3.3GHz	3.4GHz	2.8GHz	3.9GHz	3GHz	4GHz	4GHz	4GHz	3GHz
CPU Width	4	4	6	4	4	3	6	4	6	6	6	6
ROB Size	192	192	224	168	168	168	224	128	352	128	192	192
LSQ Size	72,48	72,42	72,56	64,36	64,36	44,44	72,42	48,32	128,72	64,36	64,36	72,42
L1D Cache (Size/Assoc./ Latency)	32kB/ 8way/ 4cycles	32kB/ 8way/ 4cycles	32kB/ 8way/ 4cycles	32kB/ 8way/ 4cycles	32kB/ 8way/ 4cycles	64kB/ 2way/ 4cycles	32kB/ 8way/ 5cycles	32kB/ 8way/ 4cycles	48kB/ 12way/ 5cycles	32kB/ 4way/ 4cycles	32kB/ 8way/ 4cycles	64kB/ 8way/ 5cycles
L1I Cache (Size/Assoc./ Latency)	32kB/ 8way/ 4cycles	32kB/ 8way/ 4cycles	32kB/ 8way/ 4cycles	32kB/ 8way/ 4cycles	32kB/ 8way/ 4cycles	64kB/ 2way/ 3cycles	64kB/ 4way/ 5cycles	32kB/ 8way/ 4cycles	32kB/ 8way/ 4cycles	32kB/ 4way/ 4cycles	32kB/ 8way/ 4cycles	32kB/ 8way/ 4cycles
L2 Cache (Size/Assoc./ Latency)	256kB/ 8way/ 12 cycles	256kB/ 8way/ 12 cycles	256kB/ 4way/ 12 cycles	256kB/ 8way/ 11 cycles	256kB/ 8way/ 11 cycles	512kB/ 16way/ 12 cycles	512kB/ 16way/ 17 cycles	256kB/ 8way/ 11 cycles	512kB/ 8way/ 10 cycles	256kB/ 8way/ 10 cycles	512kB/ 8way/ 8 cycles	512kB/ 16way/ 10 cycles
L3 Cache (Size/Assoc./ Latency)	1.5MB/ 16way/ 59 cycles	1.5MB/ 16way/ 36 cycles	2MB/ 16way/ 34 cycles	2MB/ 16way/ 28 cycles	2MB/ 8way/ 28 cycles	2MB/ 32way/ 40 cycles	2MB/ 16way/ 43 cycles	2MB/ 16way/ 38 cycles	2MB/ 16way/ 20 cycles	2MB/ 8way/ 20 cycles	2MB/ 16way/ 36 cycles	2MB/ 16way/ 15 cycles
RAM Size	32GB	32GB	16GB	16GB	4GB	16GB	32GB	16GB	4GB	32GB	16GB	16GB
DTLB (Num/Assoc.)	64/ 4way	64/ 4way	32/ 4way	32/ 4way	64/ 4way	48/ Full	64/ Full	32/ 4way	64/ 4way	64/ 4way	64/ 4way	32/ 4way
ITLB (Num/Assoc.)	128/ 4way	128/ 4way	32/ 4way	32/ 4way	128/ 4way	48/ Full	64/ Full	32/ 4way	64/ 4way	32/ 4way	64/ 4way	32/ 4way
STLB (Num/Assoc.)	1536/ 6way	1024/ 8way	1536/ 12way	512/ 4way	512/ 4way	512/ 4way	1024/ 8way	512/ 4way	1536/ 12way	1024/ 8way	1536/ 6way	1024/ 8way
Branch mis-penalty	20 cycles	20 cycles	20 cycles	15 cycles	14 cycles	12 cycles	19 cycles	17 cycles	14 cycles	12 cycles	12 cycles	15 cycles
Scheduler Entries	64	60	97	54	54	60	60	54	128	97	64	64

Table 3.4: Architecture Configuration Specifications

[a]Denotions used are, BW: Broadwell, HW: Haswell, SL: Skylake, SB: Sandybridge, IB: Ivybridge, K10: AMDK10, Ryz7: AMDRyzen7, NH: Nehalem, R0, R1, R2, R3 are artificial configurations

are recorded at intervals of 500,000 cycles. Counters are passed as features to predictor models in phase1 based on their correlation with the target metric in the model. Few of the commonly selected counters for the IPC metric are the total number of accesses to the L1D cache, the total number of hits in the L1D cache, the number of load accesses, load hits and load misses in the L1D cache, the number of Read For Ownership (RFO) hits in the L1D cache, the total number of hits and total number of misses in the second level cache. Few of the commonly selected counters for the AMAT metric are the total number of hits in the last level cache, total misses in the last level cache, the number of load accesses and load misses in the last level cache, the number of writeback hits and writeback misses in the second level cache, the average miss latency in the second level cache and total number of misses in the L1D cache.

3.5 Other Details on Experimental Setup

3.5.1 Microbenchmarks

In the arena of computer architecture research, performance analysis is typically performed by comparing the performance of microprocessors on widely accepted set of workload suites. These application suites that are universally deemed as an accurate representation of real-life workloads are used for performance benchmarking. For our study, workloads from the SPEC CPU2006 suite [16] have been used. The applications in these suites are generally very large and highly time-intensive for simulation. SimPoint methodology [23] [24] developed a few years ago partitions long running applications into short, orthogonal microbenchmarks. These microbenchmarks provide a fair representation of program phase behavior of each parent benchmark and is effective in completing simulation in much faster time spans. The terms microbenchmark, trace and workload are all interchangeably used to refer to the same entity through the remainder of this document.

For our purpose we use 22 traces obtained from 7 applications of the SPEC2006 suite, using SimPoint based extraction. In general, performance of an application is estimated by taking the weighted average of the performance of constituent SimPoint traces. However for our study, we treat the traces as independent microbenchmarks that represent standalone applications. Since each

trace represents a particular program phase behavior we posit this to be a valid approach.

3.5.2 Microarchitecture Simulator

The simulator used for our study is *ChampSim* [15], an in-house built lightweight simulator. Developed in collaboration by Texas A&M and Intel, the microarchitecture simulator ChampSim places focus on the memory system of microprocessor with a relatively simple core model. The choice of this simulator was due to the focus of this thesis work being on memory performance bugs; an extensive memory side detailing applicable within the simulator helps us delve deeper into exploring various memory operations and bug behaviors on these.

The simulator is configured to the architecture specification desired and run against the trace under observation for simulation. The simulation is run for a total of 10M instructions after a warmup of 1M instructions. Information in the counters is recorded every half million cycles. The simulator stores information collected from the counters and the cumulative statistics of the run in an output file at the end of each simulation.

4. PHASE 1: IPC / AMAT PREDICTOR

4.1 ML Model Design

4.1.1 LSTM

A machine learning engine used in the predictor is *LSTM (long short term memory)*. LSTM is a type of recurrent neural network first introduced by Hochreiter and Schmidhuber in 1997 [18]. It is efficient in handling sequences of data and remembering patterns in time. Conventional neural networks and machining learning techniques do not perform great on learning long term dependencies due to the vanishing gradient [25]; this method [26] combats the issue of a fast decaying gradient by back propogation that otherwise prevents training for long sequences or patterns. The output of LSTM is dependent on both the inputs and internal states. These internal cell states ensure the gradient does not decay quickly during back propogation (unfolding in time) and that information is not lost. LSTM thus makes for a popular choice in sequence prediction, text generation and purposes which require training on long sequences.

4.1.2 XGBoost

Another machine learning engine tried for the first phase of our methodology is XGBoost [19] which is the latest implementation of Gradient Boosted Trees. This employs an iterative approach where each successive stage minimizes the error of the previous stage. As a consequence, the final model built is an ensemble of results obtained from all the trees via training. The method efficiently employs boosting on decision trees while using gradient descent algorithm. XGBoost is renown for fast speed, portability and efficiency.

4.1.3 Model Setup

The LSTM machine learning models are built in Python using the deep learning library of *Keras* [27]. Keras is a high-level wrapper and uses the backend tensor framework of *TensorFlow*. LSTM takes input as a 3D tensor with shape `batch_size, timesteps, features`. The

set of samples for training is passed to the model in this format. We use the early stopping condition where loss on the validation set is observed and after 100 epochs with no improvement, the training of model is stopped. The number of neurons for each model is generally taken as $50 + int(0.5 * sequence\_length)$. Some aspects such as epochs waited on before early stopping, neurons used etc. are fine-tuned for each model to achieve the best results. The model is a 1-layer LSTM, uses Mean Squared Error as the loss function and the optimizer used in compiling the model is Adam with a learning rate of 0.01. The XGBoost ML models are also implemented in Python, using the Scikit-Learn library as a wrapper interface over the XGBoost software library. A percentage of features is sampled per tree for training the model. This avoids overfitting of the model. The number of decision trees used for gradient boosting is 200.

For the 22 microbenchmarks we have 22 independent ML models. Each model is a predictor for that microbenchmark using IPC or AMAT as the target metric. From our 24 architectures as discussed in Section 3.3, each model receives a total of 20 no-bug architectures for training the model. First, 20 bug-free samples are read in for training the model. Of these, 16 samples are used for training and 4 samples are for validation. The training of model is carried out with the 16 samples, and the validation set is only to measure the losses and signal when to stop training. From the remainder 4 no-bug architectures, 2 are used in testing the model. The other 2 architectures are Skylake designs and implemented with all the bugs to provide 22 total buggy designs. These 22 Skylake designs are also used as test samples.

The process flow till information is passed on to the second phase is as follows:

1. The simulator is configured to each architecture specification and the set of microbenchmarks is run on the simulator. The output obtained from the set of 22 architectures run without bugs and 2 architectures with bugs is parsed and stored in *csv* (comma-separated values) files.

2. The output collected from the simulator runs include the performance counters and metrics of IPC and AMAT collected at regular intervals, the cumulative metrics of IPC, AMAT and

other specific information about each simulation aggregated through the entire simulation runtime after warmup.

3. The sequences stored from performance counters and IPC/AMAT metrics of the simulator output for each trace may be of varying lengths between architectures. Since we have a single LSTM model for prediction per trace, all samples from different architectures must be made of equivalent length. The samples are pre-padded to make them equal length.

4. The 20 samples for training and validation are first read in. Depending on whether we want to predict for IPC or AMAT, we select the corresponding counter as target metric to the model. The features for the model are selected based on the correlation of performance counters with target metric. The correlation of each counter relative to target metric is checked for every sample in the training set by using the Pearson correlation coefficient. The average of this value is taken across all samples, and the counters with high correlation to the target metric are selected as features for the microbenchmark.

5. After training each model to the best extent for a workload, a set of 24 samples are tested on the predictor. Two samples are bug-free architectures while remaining have bugs. None of these architectures were seen during training.

6. We record the mean square errors and timestep errors from the IPC/AMAT prediction of the test samples relative to the original IPC/AMAT sequence obtained from simulator. These errors are then passed on to the next phase to aid in bug detection.

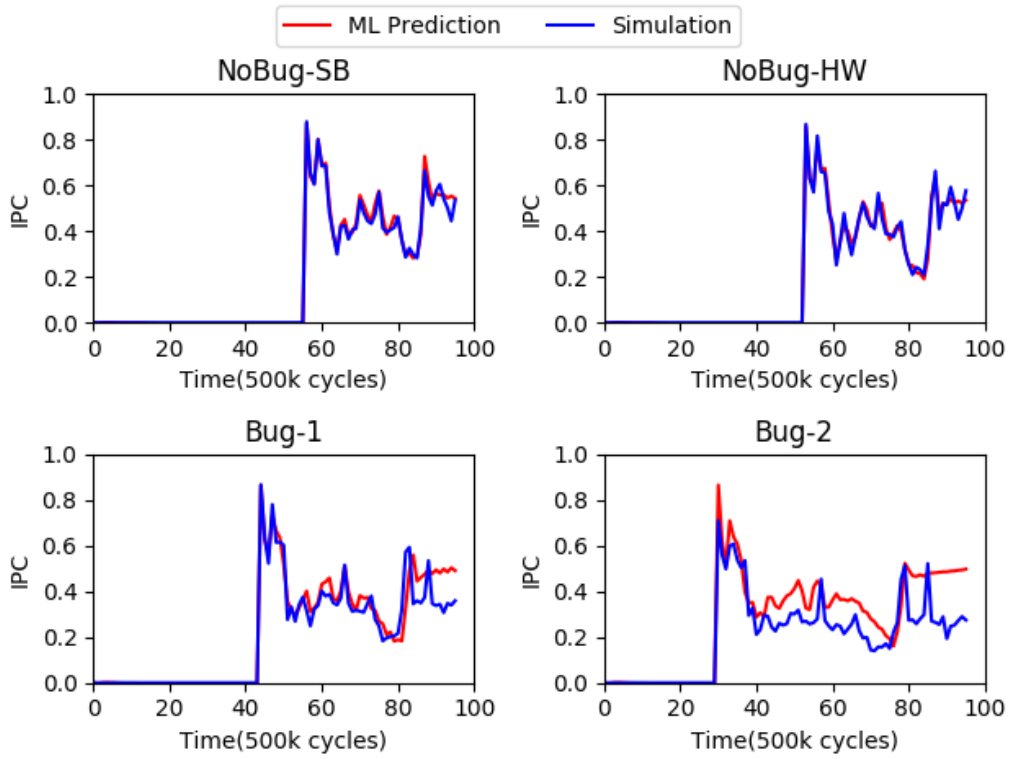

Figure 4.1: IPC Prediction Using LSTM

4.2 IPC and AMAT Predictions

4.2.1 LSTM as ML Engine

The predicted IPC and AMAT from the predictor using LSTM as the machine learning engine is shown in Figures 4.1 and 4.2. It can be observed how the prediction from the machine learning models overlap with the simulated metric for the no-bug architectures of SandyBridge and Haswell denoted by SB and HW in the figures. For the buggy architectures of Skylake implementing Bug1 and Bug2, on the other hand, there is a noticeable deviation between simulation and prediction. The figures shown are for the *perlbench41* trace.

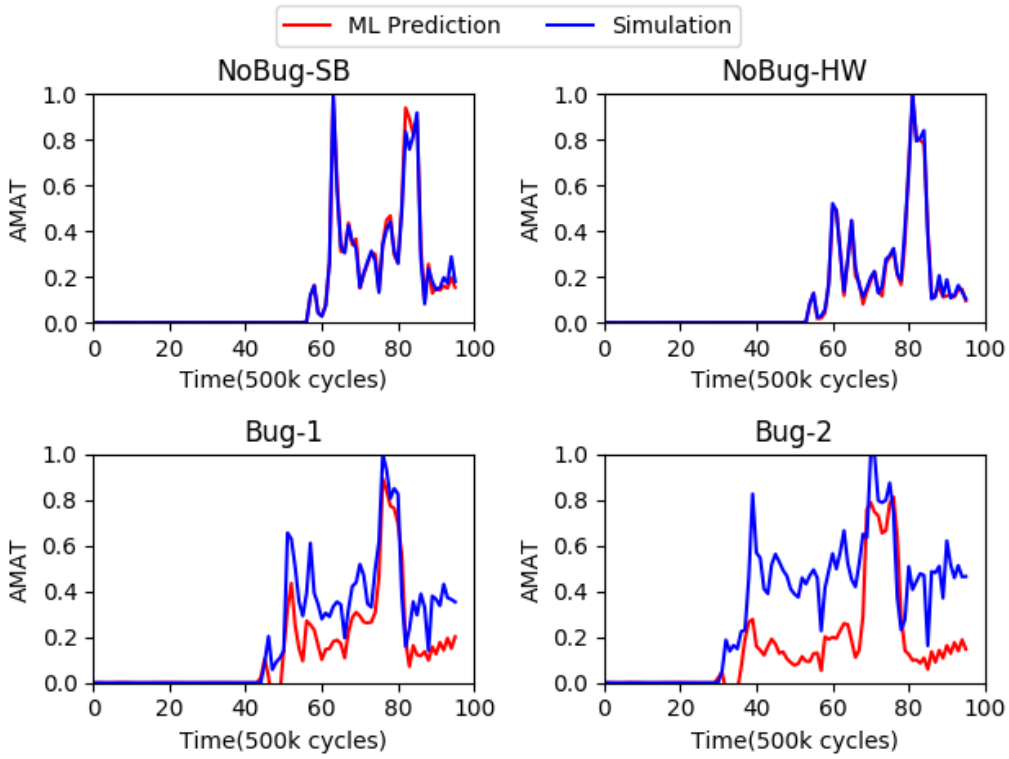

Figure 4.2: AMAT Prediction Using LSTM

4.2.2 XGBoost as ML Engine

The predicted IPC and AMAT from the predictor using XGBoost as the machine learning engine is shown in Figures 4.3 and 4.4. It can be observed that similar to the LSTM case, the prediction from the machine learning models overlap with the simulated metric for the no-bug architectures while they deviate for buggy architectures. The figures shown are for the *perlbench41* trace. The same trace has been shown for both XGBoost and LSTM predictions to exhibit how the behavior of the predictor remains similar irrespective of the ML engine used. While tuning of respective models, specific trace behavior and whether the choice of a machine learning technique would be suited for our dataset all influence the results, the aim is to present how the phase1 predictor is applicable in general and not dependent on a specific ML engine.

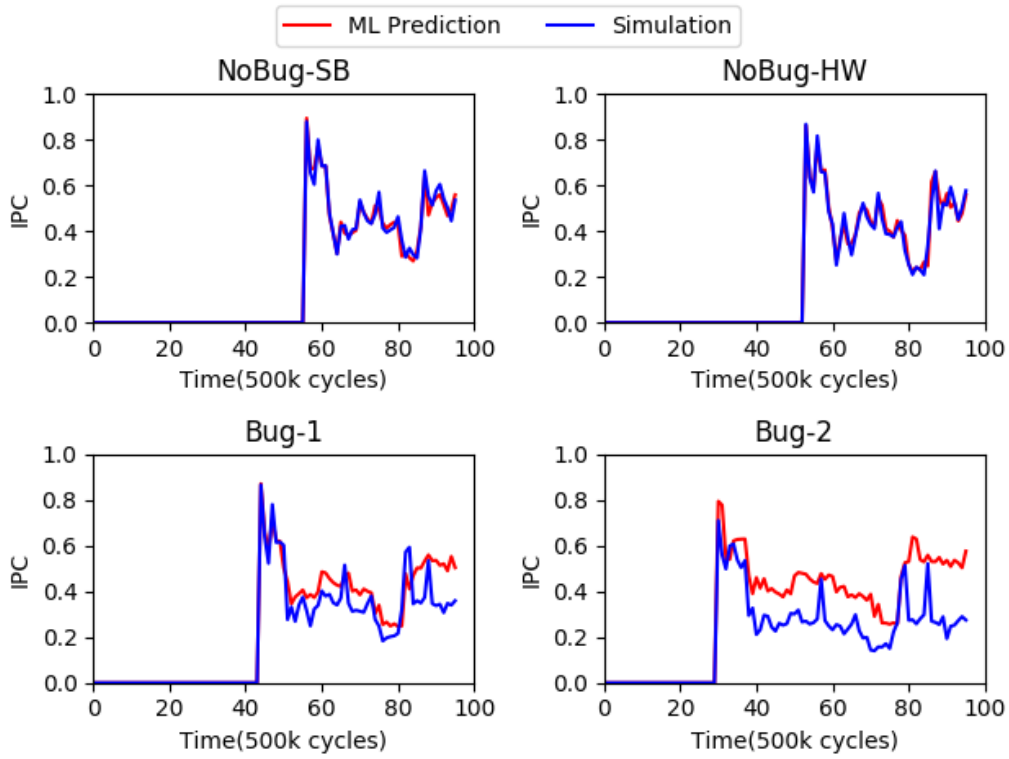

Figure 4.3: IPC Prediction Using XGBoost

4.3 Errors Recorded from the Predictor

Both **Mean Squared Error** and **Timestep Error** are recorded on the prediction of test samples. The error is calculated on the predicted IPC or AMAT from the ML predictor relative to the golden IPC or AMAT obtained from ChampSim after simulation. The IPC and AMAT metrics are data sequences here, where each data point corresponds to the value of IPC and AMAT respectively at an instant of time. **Mean Squared Error** is the mean of the squares of error between predicted and golden data points. Mean Squared Error for a single test sample hence takes the average of errors across all the data points as shown in Equation 4.1. We consider a case where there are n data points in the IPC or AMAT sequence of the test sample, Y_j refers to the actual values obtained from simulator and $\widehat{Y_j}$ denotes the predicted values.

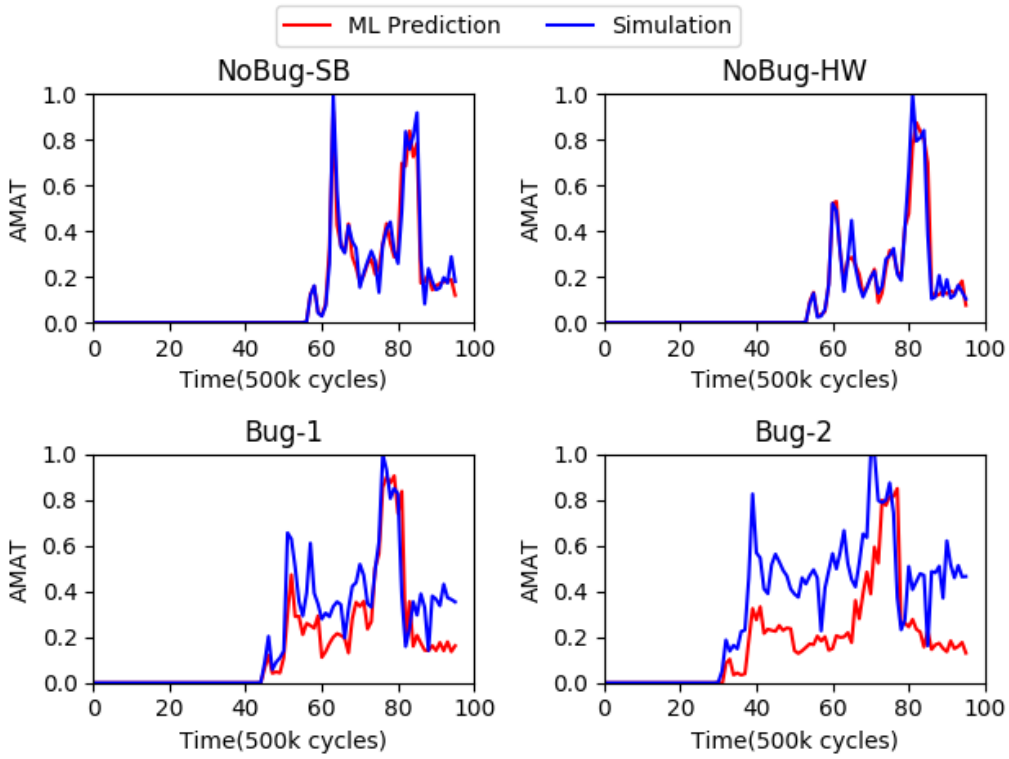

Figure 4.4: AMAT Prediction Using XGBoost

$$MSE = \frac{1}{n}\sum_{j=1}^{n}(Y_j - \widehat{Y}_j)^2 \tag{4.1}$$

Timestep Error for a single test sample or architecture design refers to the aggregate value over all the errors between predicted and golden data points for that sample. This error can be visualized as the area between the predicted and simulated curves. Equation 4.2 shows the mathematical expression for Timestep Error, where *n* denotes the total number of data points or time instants recorded, Y_j and $\widehat{Y}_j$ denote the actual and predicted values of IPC/AMAT respectively at the j^{th} timestep while Y_{j-1} and $\widehat{Y}_{j-1}$ denote the actual and predicted values respectively at the previous timestep.

$$TSE = \frac{1}{2}\sum_{j=2}^{n}(|Y_j - \widehat{Y}_j| + |Y_{j-1} - \widehat{Y}_{j-1}|) \tag{4.2}$$

Whilst most deviations from expected performance manifest as large errors and over a span of time provide a considerable Mean Squared Error (MSE) value, there might be cases where large errors appear over single or more timesteps but are overshadowed while taking the average in MSE. This is the reason for calculating Timestep Error (TSE) in addition to Mean Squared Error (MSE), it is to ensure we do not overlook these brief yet significant indication of performance anomalies. The TSE is essentially the area between the predicted and actual curves of the target metric. As seen during our experiments, this reasoning is true in many cases where the MSE may not be high but the presence of bug in the architecture is indicated by a high TSE.

A single value of MSE and TSE each is obtained from the target sequences for every test sample. These values are then passed on to the classifier to stipulate the presence or absence of a bug. Looking at both MSE and TSE offer us better odds at capturing maximum bugs.

5. PHASE 2: BUG DETECTION

5.1 Binary Classifier

For the purpose of bug detection, a simple binary classifier was developed that classifies an architecture as either with or without bugs depending on the errors collected from phase1. In this classification stage, we categorize on a per architecture basis instead of the per trace approach we followed for IPC/AMAT prediction. Each architecture is labeled as '0' indicating no bug or '1' for having a performance bug, based on certain condition checks imposed upon the collected errors.

The architecture to be classified has Mean Squared Error (MSE) and Timestep Error (TSE) values recorded for 22 traces, and there is a total of 24 test samples. So the classifer reads in 22 lists of MSE, each list with 24 values corresponding to the 24 test samples from each predictor model; and similarly for TSE. Each of the 24 test samples denotes an architecture design under test that needs to be classified as with or without bug. We apply condition checks on each of the 22 traces executed on a microarchitecture to classify that trace as buggy or bug-free. After that, voting is done on the traces to decide if the corresponding microarchitecture is to be judged as having a performance bug or not. This is why the classifier is said to operate on a per architecture basis; all the traces executed on a single microarchitecture is grouped into one category and then employed in making the final decision on whether the architecture design has a performance bug.

5.2 Classification Conditions

The classification conditions used for IPC and AMAT are similar but separate. We have two conditions each for both these assist metrics, which if satisfied then the corresponding trace for that architecture is labeled as having bug.

$$MSE \geq \eta$$
$$\text{else if} \tag{5.1}$$
$$MSE < \eta \text{ and } TSE \geq \gamma$$

In case of both metrics, for each test sample, the mean squared error is checked first. If this error is above a certain threshold (say η) then the sample is labeled '1'. If the MSE is below η then the TSE value is examined to be above a certain threshold (say γ), and if this condition is satisfied then the sample is labeled '1'. If neither of these conditions have been fulfilled then the sample is labeled '0' or bug-free. In general, the two classification conditions used for our work can be summarized in Equation 5.1. The values of the parameters η and γ are determined experientially.

The threshold values selected for IPC are determined as 0.01 for η and 0.5 for γ. For a sample with IPC as target metric to be classified as having bug either of the two conditions in Equation 5.1 must be satisfied. Failure to meet either of the two conditions would result in the sample being classified as a negative sample, meaning no bug.

In case of AMAT, η is 0.01 and γ is 0.6. For a sample with AMAT as target metric to be classified as a positive sample (having bug), the mean square error must be equal to or above 0.01. If this condition is not met, then the second condition is checked to inspect if TSE is equal to or above 0.6. If the second condition is met, then sample is marked positive. If neither of the conditions in Equation 5.1 is satisfied, then the sample is marked negative.

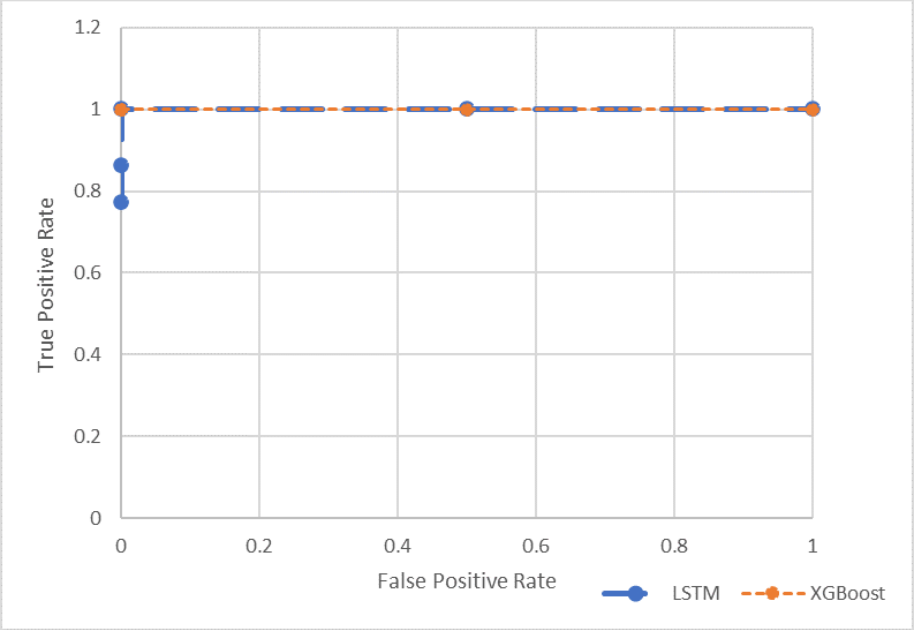

Figure 5.1: ROC Curve for IPC

The Receiver Operating Characteristic (ROC) curves for IPC and AMAT classifiers are shown in Figures 5.1 and 5.2. Since the condition on Timestep Error is the deciding factor, the value of γ is varied to plot the ROC curve for both IPC and AMAT with LSTM and XGBoost as machine learning engines. The threshold value of γ selected finally is to obtain the least false positive rate with the best possible true positive rate.

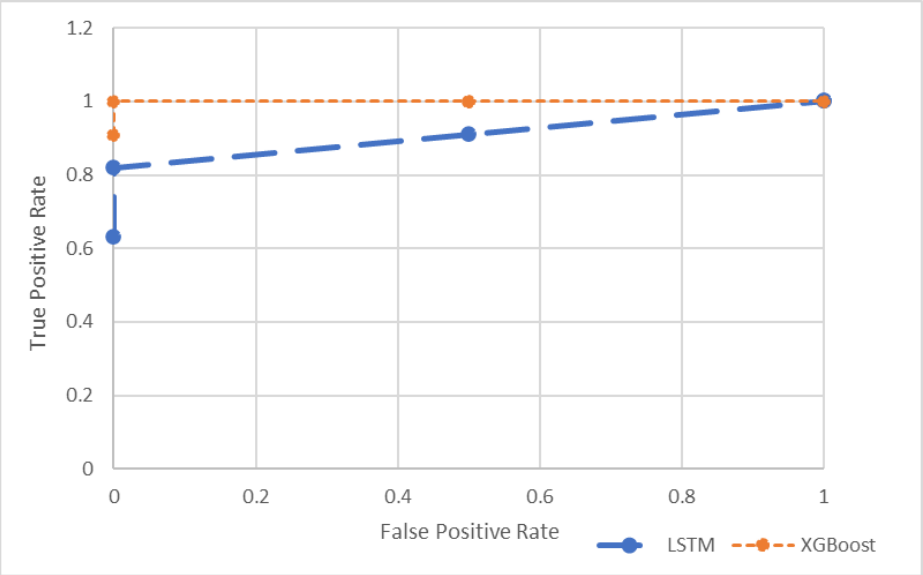

Figure 5.2: ROC Curve for AMAT

5.3 Bug Detection

After applying the classification conditions on a test sample as detailed in Section 5.2, a classification result is obtained for each workload or trace. To arrive at the final detection result we take up a voting-based approach. Suppose κ be the ratio of the number of workloads with positive classification (indicating presence of bug) to the total number of workloads used for this study. An architecture design is determined as having bug if $\kappa \geq \tau$. τ is adopted to provide maximum true positives while having zero false positives. For the IPC classifier, τ is taken as 35% of traces i.e. an architecture is judged as buggy if approximately 35% of the traces detect the design to be with bug. For the AMAT classifier, an architecture is judged as having bug if 40% of traces classify the

architecture design as an anomalous design. So, in our work, if 8 or more traces have classified the architecture design as buggy in the IPC classifer then the microarchitecture is judged as having performance bug. And for the AMAT classifier, if 9 or more traces classify the design as buggy then the architecture is judged as having bug.

6. RESULTS

6.1 Classification Results

The results obtained using our two-phase methodology are detailed in this section. Accuracy of the model, false positive rate obtained via the method, precision, recall are all among evaluation metrics used to denote how efficient a machine learning methodology is. How these evaluation metrics are calculated is outlined first, after which the values obtained for these are included in the results.

$$Accuracy = \frac{TP + TN}{P + N} \tag{6.1}$$

$$FPR = \frac{FP}{N} \tag{6.2}$$

$$Precision = \frac{TP}{TP + FP} \tag{6.3}$$

$$Recall = \frac{TP}{P} \tag{6.4}$$

The acronyms used in the equations and popular terminology are as follows :

- N and P refer to the number of actual negative and positive samples present respectively. Negative samples are those without bugs and positive samples are architecture designs which contain bugs.
- FP stands for false-positives and corresponds to negative samples that have been incorrectly judged as having bugs. FPR refers to false positive rate. FN stands for false-negatives, which is essentially an opposite to FP i.e. samples that are incorrectly labeled as without bugs.
- TP stands for true-positives and corresponds to positive samples that have been correctly judged as having bugs. TPR refers to true positive rate. TN stands for true-negatives i.e. negative samples that are correctly labeled as without bugs.

Metric	Accuracy%	FPR	Precision	Recall per Impact Bin			
				High	Medium	Low	Very Low
IPC	100	0.00	1.00	100.00	100.00	100.00	100.00
AMAT	83	0.00	1.00	100.00	100.00	100.00	20.00

Table 6.1: Classification Results Using LSTM Predictor Models

The results on bug detection from the classifier using errors obtained from the ML predictor are shown in Table 6.1 and 6.2. Formulae used to calculate the classification metrics are in Equations 6.1, 6.2, 6.3 and 6.4. Table 6.1 display classifier results using LSTM as the machine learning engine in the phase1 predictor while Table 6.2 show results for XGBoost as the machine learning engine.

Metric	Accuracy%	FPR	Precision	Recall per Impact Bin			
				High	Medium	Low	Very Low
IPC	100	0.00	1.00	100.00	100.00	100.00	100.00
AMAT	100	0.00	1.00	100.00	100.00	100.00	100.00

Table 6.2: Classification Results Using XGBoost Predictor Models

6.2 Evaluation

Classification using IPC yields 100% accuracy with both ML engines. No false positives were indicated and a precision of 1 is achieved. No bugs were left out by the classifier and all 22 bugs in the 24 test samples were correctly labeled as positive samples.

Classification using AMAT yields 83% accuracy with LSTM and 100% accuracy with XGBoost as the ML engine. No false positives were indicated and precision (also known as positive predictive value) of 1 is achieved in both cases. With XGBoost, all bugs are detected correctly using AMAT as the metric and the 2 negative architectures are also judged correctly. With LSTM, 4 bugs were missed out of the 22 bugs by the classifier. The bugs incorrectly classified by the

classifier as negative samples belong to the *very low* impact category of bugs on AMAT. 2 bugs have impacts of 1% , while the other 2 have 0.3% and 0.5% performance degradation.

It is interesting to observe that though the same metric, results from XGBoost prove better than from LSTM. While this could possibly denote a better learning and inference capability with XGBoost pertaining to our study, there are other insights as well we can explore. One may expect detection with AMAT to be more straightforward as the impact is amplified compared to corresponding bug impact on IPC generally. Among the four bugs that were missed, 2 are variants of Bug3 discussed in section 3.2 indicating a reasoning as to why the same bugs were captured by the IPC classifier using LSTM. Among the other two bugs, one of them appeared only on selected workloads and did not create any degradation for a portion of the workloads in our application suite. This behavior can be expected for some bug instances as oultined in section 1.1. These latter two bugs however had only less than 1% performance penalty and could also have been missed by the classifier owing to this very low impact. We note that there could be one of many possible reasons that a bug is missed from detection, however choice of the ML engine and assist metric also plays an important role. Both LSTM and XGBoost engines work quite well for our purpose and is suited for our data. Even the bugs missed by LSTM are very low impact bugs and in real-case these are the bugs we are less worried about since they detriment processor performance by extremely low margins.

7. CONCLUSION

The two phase methodology for automatically detecting memory performance bugs in microprocessors proposed in this thesis is a novel method that aims at fully tapping the achievable potential of a processor. By enabling automated detection of performance bugs, the technique helps us take a stride forward in utilizing all the technological advancements invested in a new processor design to the fullest extent. The methodology introduced in this thesis achieves two goals. First, it provides a reference standard for expected performance of a new processor by predicting performance using machine learning models. Second, the technique detects performance bugs on a new microarchitecture design without manual intervention or analysis. Experimental results from this work highlight that the method is effective, and a promise for further efforts in the arena of performance verification for microprocessors.

This thesis work pertains to memory performance bugs and aims at detecting them. The work has future scope of trying to localize the performance bugs and rectify the microarchitectural aspects that cause those hardware anomalies. This methodology could also be applied to the domains of processor core, interconnect and issues in multicore cache coherence to detect performance penalties and strive toward eliminating them.

REFERENCES

[1] S. Mazor, "The history of the microcomputer-invention and evolution," *Proceedings of the IEEE*, vol. 83, no. 12, pp. 1601–1608, 1995.

[2] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[3] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.

[4] S. Przybylski, M. Horowitz, and J. Hennessy, "Characteristics of performance-optimal multi-level cache hierarchies," *ACM SIGARCH Computer Architecture News*, vol. 17, no. 3, pp. 114–121, 1989.

[5] G. Hinton *et al.*, "The microarchitecture of the pentium® 4 processor," in *Intel Technology Journal*, vol. 1, 2001.

[6] D. Lin, T. Hong, Y. Li, S. Eswaran, S. Kumar, F. Fallah, N. Hakim, D. S. Gardner, and S. Mitra, "Effective post-silicon validation of system-on-chips using quick error detection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 10, pp. 1573–1590, 2014.

[7] D. Lin, E. Singh, C. Barrett, and S. Mitra, "A structured approach to post-silicon validation and debug using symbolic quick error detection," in *Proceedings of the IEEE International Test Conference*, pp. 1–10, 2015.

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

[9] J. D. McCalpin, "Hpl and dgemm performance variability on the xeon platinum 8160 processor," in *Proceedings of the International Conference for High Performance Computing,*

Networking, Storage and Analysis, pp. 225–237, 2018.

[10] 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 *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 19–33, 2019.

[11] 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. 11627–11639, 2019.

[12] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 77–88, 2012.

[13] P. Bose and J. A. Abraham, "Performance and functional verification of microprocessors," in *VLSI Design 2000. Wireless and Digital Imaging in the Millennium. Proceedings of the IEEE International Conference on VLSI Design*, pp. 58–63, 2000.

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

[15] "ChampSim: A trace based microarchitecture simulator." `https://github.com/ChampSim/ChampSim`, 2017.

[16] "SPEC CPU2006." `https://www.spec.org/cpu2006/`, 2006.

[17] J. L. Hennessy and D. A. Patterson, *Memory hierarchy design Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann Publishers, 2007.

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

[19] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM Sigkdd International Conference on Knowledge Discovery and Data Mining*, pp. 785–794, 2016.

[20] Intel.com, "7th and 8th generation Intel® core™processor family specification update," 2019.

[21] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

[22] A. Fog, "The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers." `https://www.agner.org/optimize/#manuals`, 2019.

[23] 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.

[24] 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.

[25] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *Proceedings of the International Conference On Machine Learning*, pp. 1310–1318, 2013.

[26] F. Gers, *Long short-term memory in recurrent neural networks*. Ph.D. thesis, Verlag nicht ermittelbar, 2001.

[27] F.Chollet, "Keras." `https://keras.io/`, 2015.