A COMPARATIVE APPROACH FOR UNDERSTANDING, ANALYZING, AND PREDICTING THE BEHAVIOR OF INSTRUCTION SET ARCHITECTURES AND OPERATING SYSTEMS

SHUBHANKAR SUMAN SINGH

COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY DELHI

May 2023

©Indian Institute of Technology Delhi (IITD), New Delhi - May 2023
All rights reserved.

A COMPARATIVE APPROACH FOR UNDERSTANDING, ANALYZING, AND PREDICTING THE BEHAVIOR OF INSTRUCTION SET ARCHITECTURES AND OPERATING SYSTEMS

by

SHUBHANKAR SUMAN SINGH

COMPUTER SCIENCE AND ENGINEERING

Submitted

in fulfillment of the requirements of the degree of **Doctor of Philosophy**

to the

Indian Institute of Technology Delhi

May 2023

Certificate

This is to certify that the thesis titled **A COMPARATIVE APPROACH FOR UNDERSTANDING, ANALYZING, AND PREDICTING THE BEHAVIOR OF INSTRUCTION SET ARCHITECTURES AND OPERATING SYSTEMS** being submitted by **Mr. SHUBHANKAR SUMAN SINGH** for the award of **Doctor of Philosophy** in Computer Science and Engineering is a record of bonafide work carried out by him under my guidance and supervision at the Computer Science and Engineering, Indian Institute of Technology Delhi. The work presented in this thesis has not been submitted elsewhere, either in part or full, for the award of any other degree or diploma.

Smruti Ranjan Sarangi
Professor
Dept. of Computer Science & Engg.
Indian Institute of Technology Delhi.

Acknowledgements

I would want to express my gratitude to my supervisor, Dr. Smruti R. Sarangi, who has been a very encouraging instructor and mentor throughout my entire research course. He is well regarded by all students because of his genuine concern for them and approachability, which are characteristics of a very supportive teacher. He has taught me a lot, and it is only due of his unwavering support that I am able to finish my doctorate.

I would also like to acknowledge the contribution of my parents and brother, who were supportive throughout my studies. I want to thank the support provided by all my friends and colleagues.

Lastly, I am grateful for the scholarship provided by the Prime Minister Research Fellowship scheme.

Shubhankar Suman Singh

Abstract

Modern computer systems have become fairly complex in nature; they require long development cycles and often require thousands of engineers to build and verify. In such a setting, design from scratch (ab initio design) has been replaced by a modular design process, where prebuilt components possibly sourced from third parties are used as much as possible. The role of developers is limited to system integration and coding parts of the system that are particularly novel. Given the extensive verification effort that is involved these days along with concerns about reliability and security, vendors are hesitant to make revolutionary changes in hardware/software offerings. The trend is to make evolutionary changes – small improvements with disproportionate benefits. The classical method to do such research is to thoroughly analyze a system, find and analyze bottlenecks, and then try to propose a better solution. However, the benefits of such approaches are quickly diminishing and given the scale of today's systems starting from smart watches to servers, bottom up approaches are proving to be very challenging.

Hence, we propose to look at comparative approaches where we compare the execution of two systems keeping a large part of the execution environment the same. The comparative analysis can help us pinpoint which design choices exactly lead to better performance in a set of workloads. We choose two concrete problems in this space and show that such comparative approaches yield solutions that significantly outperform the state of the art. The problems are ISA design and OS design.

We start with a quintessential problem: design an ISA for a workload with realistic constraints in mind such as minimal changes to an existing ISA. We propose a simple graphical approach that helps us nicely visualize the execution of a program. It helps us identify the issues with an ISA and also what custom instructions can be added to disproportionately increase the performance of the program. We followed up with creating a vector representation of such diagrams that includes FFTs of the equivalent 2D image and other features. We show that we can train a learner on these representations and we can then use this to perform performance estimation (for three different ISAs: x86s, ARM, and RISC-V). Such diagrams are very useful in

static single-ISA and cross-ISA performance estimation. We significantly outperform competing work in this space. We increase the performance by 10-28% of basic RISC ISAs such as RISC-V by adding 2-10 new instructions and at the same time reduce the number of infructuous and trivial suggestions made by custom ISA generators by 5-40 times. Our average error for the performance estimation is less than 2%. The prediction time of our algorithm is $5\times$ *faster than the state of the art.*

Some of this evaluation is done on an architectural simulator because it is necessary to keep the microarchitecture the same. That is when we realized that simulation technology has several shortcomings when it comes to doing such work because it does not simulate OSes correctly. We thus extended our architectural simulator Tejas to support correct and fast OS simulation. We needed to add special support for recognizing and simulating events of interest such as DVFS and DMA requests, Timer and I/O interrupts along with a fast-forward mode of simulation. The speedup in the case of our algorithm is $17\times$ *higher than SimPoint for the Linux OS. Similarly, we are* $16\times$ *and* $51\times$ *faster as compared to SimPoint for the FreeBSD and the OpenBSD OSs respectively.*

We then solved the second problem, which was automatically comparing the execution of large software on both real and simulated hardware and understanding the reasons for performance differences (if any). The motivating example here is that for the same workload, we found out that the performance difference across Linux and FreeBSD (compiled with the same libraries and compiler) can be as high as 60%. We wrote an automated tool SoftMon *that analyzes the execution of such workloads by performing extensive graph analyses, comment mining using NLP tools, and using custom heuristics. In less than 200 seconds, our tool can pinpoint a superset of 5-10 functions that are responsible for a difference in performance when a full-scale workload is run on regular operating systems: Linux, FreeBSD, and OpenBSD. We repeated the same exercise on commercial software and got many insights regarding the correct design choices for workloads. SoftMon saved 500 man hours of analysis, and found 25 reasons for 6 categories of large open-source programs.*

Comparative analyses techniques have great promise: they are vital tools for analyzing, understanding, and optimizing the performance of workloads.

सार

आधुनिक कंप्यूटर सिस्टम वास्तव में काफी जटिल हो गए हैं; उनके लिए दीर्घकालिक डेवलपमेंट साइकिल की आवश्यकता होती है और अक्सर उनके निर्माण एवं सत्यापन हेतु हजारों इंजीनियरों की आवश्यकता होती है। इस प्रकार के सेटिंग में, स्क्रैच से डिजाइन (प्रारंभिक डिजाइन) को एक मॉड्यूलर डिजाइन प्रक्रिया द्वारा प्रतिस्थापित किया गया है, जहाँ संभवतः तीसरे पक्ष से प्राप्त पूर्वनिर्मित घटकों का यथासंभव उपयोग किया जाता है। डेवलपर्स की भूमिका सिस्टम इंटेग्रेशन एवं सिस्टम के कोडिंग पार्ट्स तक सीमित है, जो विशेष रूप से नवीन है। सत्यापन के व्यापक प्रयासों, जिनमें विश्वसनीयता और सुरक्षा की चिंता भी शामिल है, के विचारगत विक्रेता हार्डवेयर/सॉफ्टवेयर पेशकशों में क्रांतिकारी बदलाव करने में संकोच कर रहे हैं। वर्तमान में छोटे-छोटे सुधारों एवं न्यूनाधिक लाभ सहित विकासपरक परिवर्तन लाने का ट्रेंड है। ऐसे शोध की शास्त्रीय विधि किसी प्रणाली के सूक्ष्मत: विश्लेषण, व्यवधानों को पहचानकर उनके विश्लेषण की है और फिर एक बेहतर समाधान सुझाने की है। हालांकि, इस प्रकार के अप्रोच के लाभ तेजी से कम हो रहे हैं और आज के स्मार्ट वाच से लेकर सर्वर तक के सभी सिस्टम्स के मामले में बॉटम अप अप्रोच चुनौतीपूर्ण साबित हो रहे हैं।

अत: हम कंपैरिटिव अप्रोच पर ध्यान देने का प्रस्ताव रखते हैं, जहां हम बृहद निष्पादन माहौल के अंतर्गत दो प्रणालियों के निष्पादन की तुलना करते हैं। तुलनात्मक विश्लेषण से हमें यह तय करने में सहयोग मिलता है कि किस डिजाइन विकल्प से वास्तव में कार्यभार के एक सेट में बेहतर निष्पादन दर्ज हो सकता है। हम यहाँ दो ठोस समस्याओं का चयन करते हैं और दिखाते हैं कि इस तरह के कंपैरिटिव अप्रोच से ऐसे समाधान निकालते हैं, जिससे अत्यंत महत्वपूर्ण निष्पादन दर्ज हो। समस्याएं आईएसए डिजाइन और ओएस डिजाइन हैं।

हम एक सर्वोत्कृष्ट समस्या से शुरू करते हैं: रियलिस्टिक कंस्ट्रेंट को ध्यान में रखते हुए वर्कलोड के लिए मौजूदा आईएसए में न्यूनतम परिवर्तन करके एक आईएसए का डिजाइन बनायें। हम एक सरल ग्राफिकल एप्रोच का प्रस्ताव रखते हैं, जो हमें एक कार्यक्रम के निष्पादन की सुंदर परिकल्पना में सहयोग दे। यह हमें आईएसए के साथ मुद्दों की पहचान करने में सहयोग देता है और यह भी सुझाता है कि कार्यक्रम के निष्पादन को और सुधारने में उसमें कौन से कस्टम इंस्ट्रक्शंस जोड़े जा सकते हैं। हमने ऐसे डयाग्रम का, जिसमें समकक्ष 2 डी इमेज की एफएफटी एवं अन्य विशेषताएँ शामिल हैं, एक वेक्टर प्रेसेंटेशन बनाते हुए उनका अनुपालन किया। हम दिखाते हैं कि हम इन रेप्रेजेंटेशनों पर एक शिक्षार्थी को प्रशिक्षित कर सकते हैं और फिर हम इसका उपयोग निष्पादन आकलन हेतु कर सकते हैं (तीन अलग-अलग आईएसए के लिए: एक्स 86, एआरएम और आरआईएससी-V)। इस तरह के डिजाइन स्टैटिक सिंगल-आईएसए और क्रॉस-आईएसए निष्पादन आकलन में बहुत उपयोगी हैं। हम आरआईएससी-V जैसे बेसिक आरआईएससी आईएसए में 2-10

नए निर्देशों को जोड़ते हुए उसका निष्पादन 10-28% तक बढ़ाते हैं और साथ ही कस्टम आईएसए जनरेटर्स द्वारा दिये गये अनुदेशों एवं ट्रिवियल सुझावों को 5-40 गुने तक कम करते हैं।

कुछ हद तक यह मूल्यांकन आर्किटेक्चरल सिमुलेटर के आधार पर किया जाता है, क्योंकि माइक्रोआर्किटेक्चर को जैसे का वैसा रखना आवश्यक है। तब हमने महसूस किया कि सिमुलेशन टेक्नॉनॉजी में कई कमियां हैं, क्योंकि इस तरह के कार्य में सिमुलेशन टेक्नॉलॉजी ओएसई का सही ढंग से अनुकरण नहीं करता। इस प्रकार हमने सही और तेज ओएस सिमुलेशन को बढ़ावा देने हेतु अपने आर्किटेक्चरल सिमुलेटर तेजस का उपयोग बढ़ाया। हमें डीवीएफएस और डीएमए अनुरोधों, टाइमर और आई/ओ इंटरप्ट्स जैसे अभिरुचिपूर्ण घटनाक्रम को पहचानने एवं उनके अनुकरण में स्पेशल सपोर्ट जोड़ने की आवश्यकता थी। हमारे एल्गोरिथ्म के मामले में स्पीडअप लाइनक्स ओएस के सिमप्वाइंट से 17× अधिक है। इसी तरह, हम फ्रीबीएसडी और ओपनबीएसडी ओएस के सिमप्वाइंट की तुलना में क्रमशः 16× और 51× आगे हैं।

फिर हमने दूसरी समस्या का हल निकाला, जो स्वचालित रूप से रियल और सिमुलेटेड हार्डवेयर दोनों के आधार पर बृहद सॉफ़्टवेयर के निष्पादन की तुलना कर रहा था और निष्पादन में अंतर (यदि कोई हो) के कारणों को समझ रहा था। यहां उल्लेखनीय उदाहरण यह है कि एक ही वर्कलोड के मामले में हमने पाया कि लाइनक्स और फ्रीबीएसडी (एक ही प्रकार के लाइब्रेरी व कंपाइलर से संकलित) के निष्पादन में 60% तक अंतर हो सकता है। हमने एक आटोमेटेड टूल सॉफ्टमोन तैयार किया, जो एक्स्टेंसिव ग्राफ विश्लेषण के निष्पादन, एनएलपी टूल का उपयोग करते हुए कमेंट माइनिंग और कस्टम हेरिस्टिक्स के उपयोग के माध्यम से ऐसे वर्कलोड के निष्पादन का विश्लेषण करता है। 200 सेकंड से भी कम समय में, हमारा टूल 5-10 फंक्शंस के एक सुपरसेट को इंगित कर सकता है, जो रेगुलर ऑपरेटिंग सिस्टम, लाइनक्स, फ्रीबीएसडी और ओपेनबीएसडी पर पूरे पैमाने पर वर्कलोड चलता है तो निष्पादन में अंतर के लिए जिम्मेदार होते हैं। हमने कमर्शियल सॉफ्टवेयर पर यही अभ्यास दोहराया और वर्कलोड के सही डिजाइन विकल्पों के बारे में अनेक इंसाइट्स प्राप्त किये। सॉफ्टमोन ने विश्लेषण के 500 मानव घंटे बचाए, और बृहद ओपेन-सोर्स प्रोग्राम्स की 6 श्रेणियों के 25 कारण पहचाने।

तुलनात्मक विश्लेषण तकनीक बहुत ही उपयोगी हैं: वे वर्कलोड के निष्पादन के विश्लेषण, उसे समझने और वर्कलोड्स के अनुकूलतम निष्पादन के महत्वपूर्ण उपकरण हैं।

Contents

Certificate **i**

Acknowledgements **iii**

Abstract **v**

Abstract in Hindi **vii**

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**

2 ISAMod: A Tool for Designing ASIPs by Comparing Different ISAs **7**

2.1 Introduction 7

2.2 Related Work 9

2.3 Simulation Framework 10

2.4 The *ISAMod* Tool 11

2.4.1 Statistics Analyzer 12

2.4.2 Image Generator 12

2.4.3 Views 13

2.5 Results 14

2.5.1 Results-I 15

2.5.2 Results-II: libquantum 17

2.5.3 Results-III: ASIP Design 19

2.6 Conclusion 22

3 CP3: A Fast and Accurate Tool For Cross-Platform Performance Prediction 23

3.1 Introduction 23

3.2 Background 25

3.2.1 Properties of the *FFT* 25

3.3 Related Work 26

3.3.1 Performance Estimation: 26

3.4 The *CP3* Tool 27

3.4.1 Trace Collection 27

3.4.2 Filter 28

3.4.3 Code Signature 29

3.5 Nature of the Use Cases 31

3.5.1 Performance Estimation 31

3.6 Evaluation and Results 32

3.6.1 Implementation Details 32

3.6.2 Application Performance Estimation 33

3.7 Conclusion 39

4 OSSim: A Fast and Accurate Architectural Simulator for Operating Systems 41

4.1 Introduction 41

4.2 Related Work 43

4.3 Operating Systems and Benchmarks 44

4.3.1 Operating Systems 44

4.3.2 Benchmarks 46

4.4 Simulation Framework 47

4.4.1 *QemuTrace* 47

4.4.2 Tejas Simulator 50

4.4.3 Flexibility and Usability 51

4.4.4 Trace Analysis 52

4.5 Evaluation 56

4.5.1 Methodology 56

4.5.2 Accuracy of *QemuTrace*: 56

4.5.3 Statistics: DMA and Interrupts 59

4.5.4 Accuracy and Speedup of the Simulator: 59

4.5.5 Combined Workloads (HPC) (*CP3* + *OSSim*) 63

4.6 Conclusion 64

5 SoftMon: A Tool to Compare Similar Open-source Software from a Performance Perspective 65

5.1 Introduction 65

5.2 Background and Related Work 67

5.2.1 Taxonomy of Prior Work 67

5.2.2 Techniques to Find Code Similarity 70
5.2.3 Performance Analysis 71
5.3 The Design of *SoftMon* 72
5.3.1 Trace Collector 73
5.3.2 Classification and Clustering 74
5.3.3 Graph Engine 74
5.3.4 Annotation 76
5.3.5 Map Engine 77
5.3.6 Library Engine: Analysis of Stack Traces 78
5.3.7 Graph Visualization 80
5.4 Evaluation 81
5.4.1 Implementation Details 81
5.4.2 Execution of the *Graph Engine* 81
5.4.3 Execution of the *Map Engine* 84
5.4.4 Differences in Performance 84
5.4.5 Reasons for the Differences in Performance 85
5.4.6 Summary of Differences 90
5.4.7 User Study 90
5.5 Conclusion 92

6 Conclusion and Future Work 93

Bibliography 95

List of Publications **109**

Biodata **111**

List of Figures

1.1 An overview of the work done 3

2.1 Performance of ARM and RISC-V ISAs (relative to x86) 8

2.2 Simulation framework 11

2.3 The *ISAMod* tool 11

2.4 Raster diagram from an instruction sequence 13

2.5 Images for View II: x86 vs ARM vs RISC-V 14

2.6 libquantum: detailed analysis 17

2.7 Output of the FAGECI algorithm for *deepsjeng* 19

2.8 Performance gain for RISC-V: FAGECI vs *ISAMod* 21

2.9 libquantum - IPC: RISC-V=1 to RISC-V-ASIP=1.16 21

3.1 Overview 24

3.2 Overview: The ISA-MON tool 27

3.3 Signature extraction example: *deepsjeng* on RISC-V 29

3.4 Performance of RISC-V and x86 w.r.t ARM 33

3.5 Error (%) in runtime estimation for a) ARM b) RISC-V and c) x86 35

3.6 pcm (ARM): (a) Original Cycles vs Predicted Cycles (b) Probability distribution of cycles 37

3.7 Error vs Instruction Window Size 38

3.8 Correlation of Sampled vs Estimates (IPC) 38

4.1 Simulation framework 47

4.2 Task Analysis: Phase Detection 52

4.3 Task Analysis: Classification 52

4.4 Task Analysis 53

4.5 Linux: a) fsdisk-r b) syscall-m 55

4.6 Contribution of system calls: hardware vs *QemuTrace* 58

4.7 *OSSim*: (a) Error in IPC (b) Branch MPKI (c) L1 MPKI (d) L2 MPKI 61

4.8 The OS and application components 62

4.9 Flamegraph of the *process_vm_readv* system call on Linux 62

4.10 Percentage error in IPC 62

5.1 Prior work - classification 69

5.2 Flow of actions 72

5.3 Graph Engine: (a) Initial (b) Reduce (c) Filter (d) Partition (*getdents* system call for *Find* in Linux) 73

5.4 Mapping Engine pairs: {(A1, A2); (A2, B2); (A3, B3)} 78

5.5 Breakup of Cycles (smaller value is better): (a) Image (b) Pdf (c) Text (d) Audio (e) Mail (f) Operating system 83

5.6 Case Studies: Output of *SoftMon* (a) Image Tools (b) Pdf readers (c) Operating Systems 85

5.5 *SoftMon*: runtime of different steps 82

5.6 Summary of differences in Applications (**B/F** → **Performance Bug/Feature**) . 87

5.7 Summary of differences in OSs (✔ : Optimization available, ✖ : Optimization not available, – : Not relevant) 88

List of Tables

2.1 Number of new custom instructions 20

3.1 Code Signature $\mathscr{S}_I$ 30

3.2 System parameters 33

4.1 Summary of Prior Work 43

4.2 Operating systems and their versions 46

4.3 Tool chain used to compile the benchmarks 46

4.4 Virtual Machine Configuration 50

4.5 Simulation parameters 56

4.6 Number of cycles taken for a DMA operation (fsdisk-r benchmark on Linux) . 59

4.7 Number of cycles taken between consecutive DMA requests (fsdisk-r benchmark on Linux) 59

4.8 Speedup and error: SimPoint vs *OSSim* 60

5.1 Popular open-source software categories 66

5.2 Summary of Prior Work 68

5.3 Software Categories, Properties and Description of Benchmarks (*loc* $\rightarrow$ lines of code) 80

5.4 Comment Similarity (from *spaCy*) 82