NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Assessment of modern RISC-V microprocessors reliability using runtime hardware measurements

Ilias P. Konstantinidis

Supervisor: **Dimitris Gizopoulos,** Professor

ATHENS

APRIL 2025

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Εκτίμηση της αξιοπιστίας των σύγχρονων επεξεργαστών RISC-V με χρήση μετρήσεων υλικού κατά τον χρόνο εκτέλεσης

Ηλίας Π. Κωνσταντινίδης

Επιβλέπων: **Δημήτριος Γκιζόπουλος,** Καθηγητής

ΑΘΗΝΑ

ΑΠΡΙΛΙΟΣ 2025

BSc THESIS

Assessment of modern RISC-V microprocessors reliability using runtime hardware measurements

Ilias P. Konstantinidis

S.N.: 1115202000109

SUPERVISOR: **Dimitris Gizopoulos,** Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Εκτίμηση της αξιοπιστίας των σύγχρονων επεξεργαστών RISC-V με χρήση μετρήσεων υλικού κατά τον χρόνο εκτέλεσης

Ηλίας Π. Κωνσταντινίδης

Α.Μ.: 1115202000109

ΕΠΙΒΛΕΠΩΝ: Δημήτριος Γκιζόπουλος, Καθηγητής

ABSTRACT

Assessing hardware reliability against different external or internal disturbances is a critical challenge in processor design, especially in the context of complex microarchitectures with out-of-order (O3) execution, where increased instruction-level parallelism can differentiate the impact of transient faults. This thesis explores the prediction of the Architectural Vulnerability Factor (AVF - the standard metric for transient faults measurements) and related error outcomes (Silent Data Coruptions - SDCs, Timeouts, Assertions/Crashes) across key hardware structures (Register File, L1 Data Cache, L1 Instruction Cache) of designs with a RISC-V architecture. Utilizing the popular gem5 simulator, a series of automated Python scripts were developed to create and execute checkpoints for the collection of runtime hardware metrics. A recent microarchitectural modeling and injection framework (gem5-MARVEL) was employed to calculate the corresponding AVF values through statistically injecting single-bit faults into random locations of the hardware structures and CPU cycles during program execution. Feature selection strategies based on the correlation of the performance metrics were implemented to identify the most relevant hardware metrics for each component and support efficient regression procedures. Several regression techniques (linear, polynomial, ridge and lasso models), with additional analysis performed using the Patient Rule Induction Method (PRIM), were evaluated using various scientific Python libraries. While moderately strong R^2 values were observed in the case of total-AVF and SDC-AVF, our final conclusions highlight difficulties in accurate AVF prediction at runtime.

SUBJECT AREA: Hardware Reliability

KEYWORDS: Architectural Vulnerability Factor (AVF), Silent Data Corruption (SDC), fault injections, AVF estimation, regression

ΠΕΡΙΛΗΨΗ

Ο υπολογισμός της αξιοπιστίας του υλικού ενάντια σε διαφορετικές εξωτερικές ή εσωτερικές διαταραχές αποτελεί μια μεγάλη πρόκληση στο σχεδιασμό επεξεργαστή (CPU design), ειδικά στο πλαίσιο σύνθετων μικροαρχιτεκτονικών με εκτέλεση εκτός σειράς (out-of-order, O3) όπου ο αυξημένος παραλληλισμός σε επίπεδο εντολών μπορεί να διαφοροποιήσει την επίδραση των παροδικών σφαλμάτων (transient faults). Αυτή η διατριβή διερευνά την πρόβλεψη του Συντελεστή Αρχιτεκτονικής Ευπάθειας (Architectural Vulnerability Factor, AVF) και των υπολοίπων σχετικών εσφαλμένων αποτελεσμάτων (Silent Data Coruptions - SDCs, Timeouts, Assertions/Crashes) σε σημαντικές δομές του επεξαργαστή (Αρχείο Καταχωρητών, Κρυφή Μνήμη Δεδομένων πρώτου επιπέδου, Κρύφη Μνήμη Εντολών πρώτου επιπέδου) που σχεδιάστηκαν στα πλαίσια την αρχιτεκτονικής RISC-V. Χρησιμοποιώντας τον διάσημο προσομοιωτή gem5, αναπτύχθηκε μια σειρά από αυτοματοποιημένα Python scripts για την δημιουργία και την εκτέλεση σημείων ελέγχου (checkpoints) και την συλλογή μετρήσεων υλικού κατά τον χρόνο εκτέλεσης. Ένα σύγχρονο πλαίσιο μικροαρχιτεκτονικής μοντελοποίησης και εισαγωγής ελαττωμάτων (gem5-MARVEL) χρησιμοποιήθηκε για τον υπολογισμό των αντίστοιχων τιμών AVF μέσω της στατιστικής εισαγωγής ελαττωμάτων ενός bit σε τυχαίες θέσεις και κύκλους CPU κατά την εκτέλεση του προγράμματος. Διάφορες στρατηγικές επιλογής χαρακτηριστικών που βασίζονται στη συσχέτιση των μετρήσεων απόδοσης εφαρμόστηκαν για τον εντοπισμό των πιο σχετικών μετρήσεων για κάθε δομή επεξεργαστή και την εφαρμογή αποτελεσματικών διαδικασιών παλινδρόμησης (regression procedures). Αρκετές τεχνικές παλινδρόμησης (γραμμικά, πολυωνυμικά, ridge και lasso μοντέλα), με επιπλέον ανάλυση που πραγματοποιήθηκε χρησιμοποιώντας τη μέθοδο PRIM (Patient Rule Indction Method), αξιολογήθηκαν χρησιμοποιώντας διάφορες επιστημονικές βιβλιοθήκες της Python. Ενώ παρατηρήθηκαν μέτρια ισχυρές τιμές R^2 στην περίπτωση του συνολικού AVF και SDC-AVF, τα τελικά αποτελέσματα φανερώνουν δυσκολίες στην αξιόπιστη πρόβλεψη του AVF κατά τον χρόνο εκτέλεσης.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Αξιοπιστία Υλικού

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Συντελεστής αρχιτεκτονικής ευπάθειας, Σιωπηλά σφάλματα, εισαγωγή ελαττωμάτων, εκτίμηση AVF, παλινδρόμηση

ACKNOWLEDGEMENTS

I would like to thank my professor, Dimitris Gizopoulos, the postdoctoral researcher, George Papadimitriou, and the PhD student, Foteini Kotsimpou, for their help and support during this thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Importance of Hardware Reliability

Reliable computer systems are essential for ensuring performance, efficiency, and security in various fields, including semiconductors, finance, communication and healthcare. A reliable system can prevent data loss, minimizes error margin and ensures operational stability and consistent performance.

Computers are heavily dependent on hardware reliability, as even the most advanced software cannot function properly without stable and durable hardware components. Hardware failures, such as malfunctioning processors, faulty memory, or failing storage devices, can lead to system crashes, data corruption, and operational disruptions. Hardware reliability refers to the ability of a computing system to function correctly over time without failure. The measurement of how well a hardware component (or a full computing system) can withstand environmental conditions, such as cosmic rays or radiation[6], while maintaining performance and correctness constitutes an important issue in the field of computer engineering.

The importance of hardware reliability stems from the need of proper function of multiple applications and systems in today's world:

- **Integrity of Data Centers**: Unreliable hardware leads to increased maintenance costs, downtime, and potential data loss. Data centers and cloud computing providers invest heavily in reliable hardware and vulnerability research to ensure continuous service and avoid revenue loss.
- **Environmental and Economic Impact**: Unreliable components can lead to performance degradation, requiring frequent maintenance or replacements. On the other hand, high reliability reduces electronic waste and the need for frequent replacements, contributing to environmental sustainability and cost savings.
- **Safety in Critical Systems**: In industries such as aerospace, healthcare and automotive, hardware failures can lead to catastrophic consequences. For example, avionics systems in aircraft or life-supporting medical devices must function flawlessly to avoid life-threatening situations.

1.2 Quantifying Hardware Vulnerability

Reliability is often quantified using metrics like Failure In Time (FIT). FIT is a metric used to express the failure rate of hardware components, representing the number of failures expected to occur per 10^9 hours of operation. For example, a component with a 10 FIT rating is predicted to experience 10 failures over one billion hours of use. FIT values can be determined using reliability metrics, specifically the Mean Time Between Failures (MTBF) or Mean Time To Failure (MTTF). MTBF represents the average operational time between two consecutive failures of a repairable system, while MTTF measures the expected lifetime of a non-repairable system or component before it fails. Lastly, Mean Time To Repair (MTTR) is the average time required to diagnose, repair, and restore a failed system back to operation.

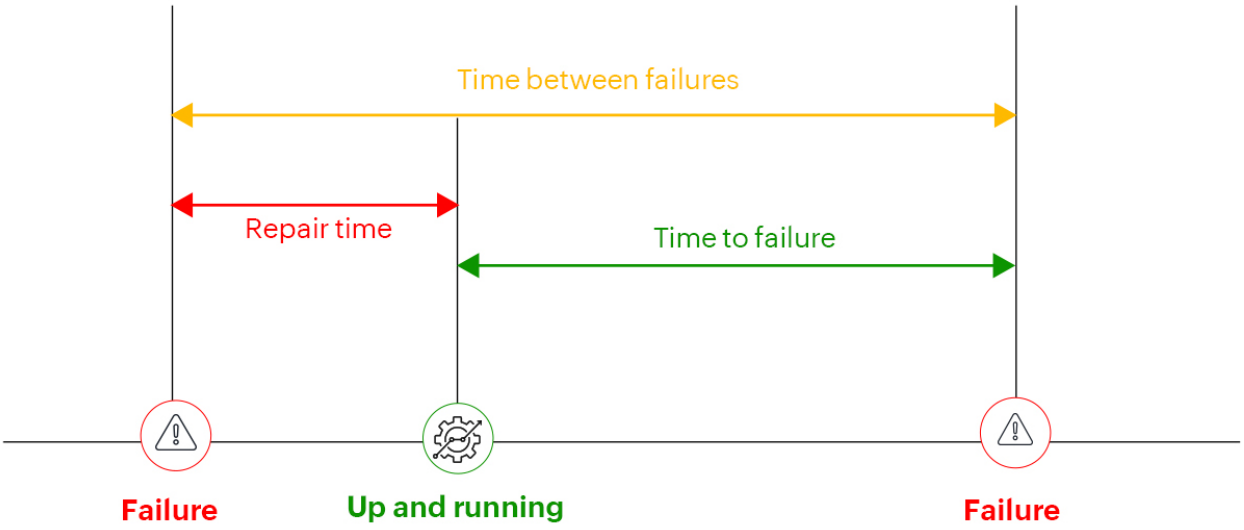

Figure 1.1: MTBF, MTTF and MTTR

Another widely adopted reliability metric is the Architectural Vulnerability Factor (AVF), which measures the likelihood that a hardware fault will lead to an actual system failure[20]. Usually, the AVF is estimated or calculated for each individual hardware structure of a CPU or other compute chip. Not all faults cause functional errors in workload execution, so AVF helps determine how susceptible a system is to failures caused by soft errors, such as cosmic radiation or electrical disturbances. Understanding and accurately measuring AVF is essential for designing resilient systems, optimizing fault-tolerant hardware, and ensuring high availability in critical applications.

1.3 Motivation

This thesis aims to develop a statistical regression-based model for dynamically predicting the Architectural Vulnerability Factor (AVF) at runtime, using hardware proxies derived from microarchitectural features. The main motivation factors of this research are:

- Conventional AVF analysis, such as Architecturally Correct Execution (ACE)[20], extensive simulations or fault injection experiments, are computationally expensive and fail to capture real-time variations. By introducing machine learning regression techniques, a fast, lightweight and accurate real-time AVF estimation approach for different hardware components may be built.
- Knowing the AVF in real-time enables the processor to apply dynamic redundant multi-threading (RMT) or other mitigation techniques. RMT is the technique of running two copies of the same program as separate threads, feeding them identical inputs, and comparing their outputs[17]. Accurate AVF runtime estimation allows

for a threshold to be set that enables and disables RMT based on the desirable performance and reliability of the specific application.

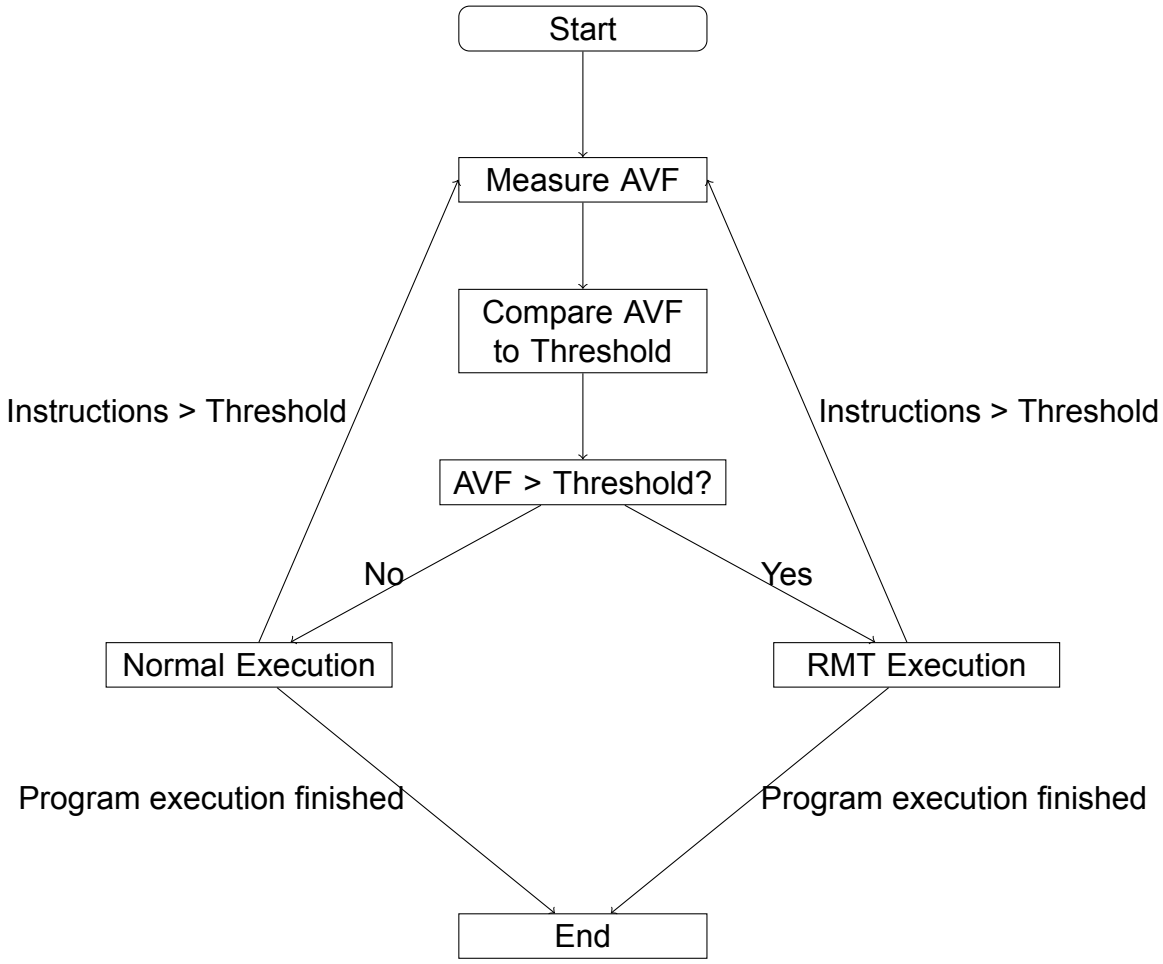

Figure 1.2: Dynamic Redundant Multi-Threading (RMT) based on AVF Estimation

- In applications such as aerospace, automotive, and medical devices, real-time AVF estimation ensures that processors can respond to soft errors before they cause failures. This can prevent catastrophic consequences in mission-critical operations.

1.4 Related Works

The concept of Architectural Vulnerability Factor (AVF) was initially introduced by Mukherjee et al.[20]. In the same research an approach of AVF estimation was proposed called Architecturally Correct Execution (ACE). The establishment of the AVF metric in the field of research created the need for the development of a unified framework for architectural level software reliability analysis. Sim-SODA[10] allowed for the first regression-based approaches of AVF prediction to be explored[21][15].

Although Sim-SODA and other frameworks are still powerful tools for hardware reliability analysis, they were limited in terms of ISAs and general configurations. Also, although fast, ACE is a conservative method for calculating AVF, providing a pessimistic upper bound rather than an accurate estimate of vulnerability. These limitations, coupled with the rise in popularity of the RISC-V architecture[22] led to the formation of a highly-configurable and accurate infrastructure[8][19][18][13] that uses statistical fault injection

(SFI)[14] for the computation of AVF. This newly emerged groundwork brings about the reexamination of the previous prediction approaches and opens new possibilities for research exploration. In this thesis, both existing and new methods regarding runtime AVF prediction are tested using gem5 and gem5-MARVEL in RISC-V architecture.

2. BACKGROUND

2.1 Faults and Errors Terminology

Understanding the definitions of the core concepts in vulnerability analysis, as well as the different types of errors and their causes in modern CPUs is an essential first step in calculating and improving the reliability of the system.

The terms defects, faults, and errors are often used interchangeably in various domains, including hardware design, software engineering, and system reliability analysis. However, they have distinct meanings, and misunderstanding these terms can cause confusion in diagnosing and resolving issues[4].

- **Defect**: A defect is the physical cause of the problem or in other words an inherent flaw in a system's design, manufacturing, or material composition.
- **Fault**: A fault is the actual manifestation of a system defect. It represents a model that may cause deviations from expected behavior. Faults are useful to better understand the behavior of a system under an infinite set of physical defects and also a systematic means to calculate the effectiveness of methods to detect and mitigate defects
- **Error**: An error is the observable incorrect behavior that results from a fault (and the defect it models). Errors are what users or monitoring systems detect as failures in the system's operation.

As an example, a CPU with a microscopic impurity (defect), may have a faulty transistor that is stuck-short (fault), which oftentimes causes incorrect arithmetic calculations at the software level (error).

In simulated environments, when it comes to hardware reliability, faults and errors are the focus of attention and, therefore, it is essential to understand their different types.

- **Transient faults**: Transient faults occur temporarily and they remain in the system state until the affected data is overwritten. They are often caused by environmental factors, such as cosmic radiation[6], electromagnetic interference, or voltage fluctuations. Errors caused by radiation or electrical disturbances are also known as soft errors.
- **Intermittent faults**: Intermittent faults occur sporadically and unpredictably, often due to unstable physical connection or components that operate near their tolerance limits.
- **Permanent errors**: Permanent errors remain in the system until a repair or replacement is performed. They usually result from physical damage, wear and tear, or irreversible defects in materials.

Transient faults is the type of faults this thesis deals with, as discussed later.

2.2 Architectural Vulnerability Factor

The Architectural Vulnerability Factor (AVF) quantifies how likely transient hardware fault will result in an observable wrong output or to a system failure. AVF is widely used in reliability analysis to assess the susceptibility of various hardware components to soft errors caused by transient faults. However, not all faults lead to visible errors. Many faults are masked have no impact on program correctness. Here are some categories of masked faults:

- **Microarchitectural Masking**: Faults that occur in unused processor states or idle execution units, making them irrelevant.
- **Logical Masking**: Fault occurs in a part of the circuit that does not affect the final computation[20]. These faults could affect the control flow of a program execution and therefore its performance.
- **Algorithmic Masking**: Certain software-level computations naturally correct or ignore incorrect values, preventing faults from propagating.

Except masked faults, a transient fault leads to one of the following outcomes:

- **Silent Data Corruption (SDC)**: Program execution was completed, but wrong output was produced[12] without any observable notification
- **Crash/False Assertion**: Program execution was terminated unexpectedly.
- **Timeout**: Program execution took longer to complete than the allowed duration. Most likely the execution was stuck in an infinite loop.

2.3 Vulnerability Analysis Techniques

Vulnerability analysis techniques help assess the resilience of a system against faults and errors. The available techniques can be broadly classified into static and dynamic analysis. Static analysis examines the system at design time, analyzing the hardware description or software code without executing it. In contrast, dynamic analysis evaluates the system during execution. The most well-known static and dynamic analysis techniques for AVF estimation are ACE[20] and SFI[14] respectively.

2.3.1 Architectural Correct Execution

Architectural Vulnerability Factor (ACE) is a static reliability analysis technique used to estimate a system's Architectural Vulnerability Factor (AVF) by determining which bits are essential for correct execution (ACE bits) and which can tolerate faults without affecting program correctness (Un-ACE or Masked bits)[20]. In short, a bit is classified as ACE if it satisfies certain conditions, such as control flow dependency and memory addressing impact.

As an example, the diagram shown in 2.1 represents an 16-bit register, where after performing ACE analysis, some bits are classified as ACE bits (red) (faults in these bits

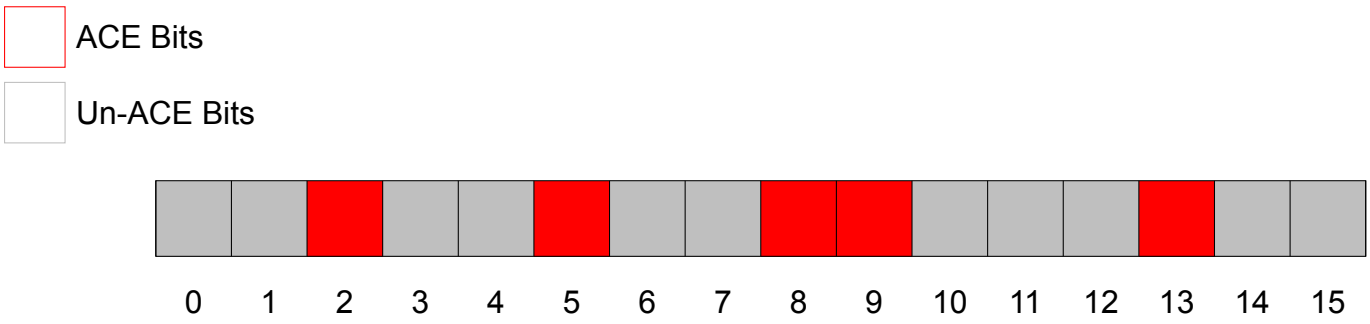

Figure 2.1: A register with ACE (red) and Un-ACE (gray) bits

affect program correctness) and other bits are Un-ACE bits (gray) (faults in these bits do not impact execution).

The total-AVF is the fraction of bits that are ACE, meaning they contribute to system failures when faults occur. The formula is:

$$AVF_{total} = \frac{ACE\ Bits}{Total\ Bits} \tag{2.1}$$

Substituting the values:

$$AVF_{total} = \frac{5}{16} = 0.3125\ (31.25\%)$$

This means that 31.25% of faults in this register could impact program execution, while 68.75% of faults would be masked.

2.3.2 Statistical Fault Injection

gem5-MARVEL uses Statistical Fault Injection (SFI) for calculating AVF. SFI is a technique used to evaluate the reliability of a hardware component by single injecting a fault in the component's values for a number of program executions and observing their effects. Instead of testing every possible fault, SFI randomly samples fault location and times, making it a scalable and efficient statistical approach. For example, to evaluate the reliability of register file in a CPU, the SFI approach is the following:

1. Randomly select a register, a bit position, and a time step during execution.
2. Flip a randomly chosen bit in the selected register (e.g. changing a 0 to 1 or vice versa).
3. Continue execution and monitor the system for different outcomes (Masked Faults, SDCs, Timeouts, Assertions/Crashes)
4. Repeat steps 1-3 for N number of times.
5. Calculate total-AVF using the following formula:

$$AVF_{total} = \frac{Number\ of\ Faults\ Leading\ to\ Errors}{Total\ Number\ of\ Faults} \tag{2.2}$$

This formula, based on the possible program outcomes, expands to:

$$AVF_{total} = \frac{AVF_{SDC} + AVF_{Timeout} + AVF_{Assertion/Crash}}{Total\ Number\ of\ Faults} \tag{2.3}$$

Each category has unique behavior and are worth trying to estimate separately. Thankfully, gem5-MARVEL provides the ability to distinguish these categories for each fault injection. Finally, SFI's margins of error have been quantified and the number of injections required in order to achieve a desired confidence can be calculated[14].

For example, after performing SFI in the register file for N = 100, on a random program execution, the results were:

- **Masked**: 71
- **SDCs**: 20
- **Assertions/Crashes**: 5
- **Timeouts**: 4

Substituting in 2.3, the total-AVF is calculated as:

$$AVF_{total} = \frac{20 + 5 + 4}{100} = \frac{29}{100} = 0.29\ (29\%)$$

In 2.2, the main differences between SFI and ACE are presented. In general, ACE is considered a faster but more conservative method, while SFI is time-consuming but much more accurate. Also, SFI can be configured, since it allows specifying the number of injected executions based on the desired margin of error.

2.4 Instruction Set Architectures

An Instruction Set Architecture (ISA) defines the interface and communication between software and hardware. It specifies the behavior of the hardware for every given CPU instruction by providing an encoding for every one of them that belongs to the ISA.

Every ISA is unique and has its own characteristics that impact processor performance, power efficiency, complexity and reliability. However, over the years, two dominant paradigms have emerged: Reduced Instruction Set Computing (RISC) and Complex Instruction Set Computing (CISC).

2.4.1 RISC vs CISC

RISC architectures simplify the processor by implementing only a small set of simple instructions that are frequently used while less common operations are implemented as subroutines. The instructions usually execute in one single cycle and have fixed length. Examples of RISC architectures include RISC-V, ARM, PowerPC, and MIPS.

CISC architectures use a large set of complex instructions, some of which can execute multiple low-level operations on a single instruction. They have variable instruction length

Aspect	Statistical Fault Injection (SFI)	Architecturally Correct Execution (ACE)
Methodology	Simulates faults and observes their propagation in the system.	Assumes faults are masked or do not cause visible failures due to redundancy.
Accuracy	When performed with high statistical significance, provides an accurate estimate of system vulnerability through fault propagation.	Provides an upper bound estimate, which is conservative and less accurate (in many cases a severe overestimation of the actual vulnerability).
Computational Cost	High computational overhead due to multiple fault injection simulations.	Low computational cost, fast and efficient.
Purpose	Detailed empirical analysis of fault propagation and system vulnerability.	Quick approximation of the system's worst-case fault vulnerability.
Use Case	Used for high-accuracy reliability analysis and detailed fault behavior modeling.	Useful for early-stage design to estimate potential vulnerabilities quickly.

Table 2.1: Comparison of Statistical Fault Injection and Architecturally Correct Execution

Aspect	CISC	RISC
Origin	The original microprocessor ISA.	Redesigned ISA that emerged in the early 1980s.
Clock Cycles per Instruction	Instructions can take several clock cycles.	Single-cycle instructions.
Design	Hardware-centric design; the ISA does as much as possible using hardware circuitry.	Software-centric design; high-level compilers take on most of the burden of coding many software steps from the programmer.
RAM Usage	More efficient use of RAM than RISC.	Heavy use of RAM (can cause bottlenecks if RAM is limited).
Complexity	Complex and variable length instructions.	Simple, standardized instructions.
Layers of Instructions	May support microcode (microprogramming where instructions are treated like small programs).	Only one layer of instructions.
Number of Instructions	Large number of instructions.	Small number of fixed-length instructions.
Addressing Modes	Compound addressing modes.	Limited addressing modes.

Table 2.2: Comparison of CISC and RISC architectures

and are designed to minimize software complexity. The most well-known CISC architecture is x86 (Intel and AMD processors).

Despite their differences, not much research has been conducted in terms of reliability analysis of RISC vs. CISC architectures. Most studies tend to focus on performance, power efficiency, and scalability[11], leaving a gap in comprehensive fault tolerance evaluations.

2.4.2 RISC-V Architecture

RISC-V is an open-source, royalty-free RISC architecture that has gained significant traction in both academic research and industry[22]. Unlike proprietary ISAs, such as ARM and x86, RISC-V allows customization and modifications without license fees.

RISC-V instructions are designed to be simple, modular and extensible, following the RISC principles. The ISA is divided into the types below:

- **Integer Instructions (I)**: Basic arithmetic, logic and control instructions.
- **Multiplication and Division (M)**: Instructions for integer multiplication and division.
- **Floating-Point Instructions (F & D)**: Single (F) and double (D) precision floating-point operations.
- **Compressed Instructions (C)**: A subset of 16-bit instructions to reduce code size and improve efficiency.
- **Vector Extensions (V)**: Instructions for SIMD[3] (Single Instruction, Multiple Data) operations.
- **Privileged Instructions**: Instructions meant for system-level operations, including supervisor mode and memory management.

As RISC-V adoption grows across various fields, ensuring its reliability becomes crucial. Unlike established ISAs, RISC-V's customization ability and open-source nature introduce additional complexity and variability in reliability. A complete and comprehensive fault analysis framework like gem5-MARVEL is essential to assess vulnerabilities in the various RISC-V designs encompassing fault injections experiments, AVF evaluation and correction mechanisms.

3. WORKING ENVIRONMENT

3.1 gem5

gem5[1] is a widely used and open-source architectural simulator that allows evaluating processors designs, including RISC-V. It provides the ability to model different ISAs, memory hierarchies and microarchitectures, making it a powerful tool for performance analysis of program execution.

gem5 has two primary modes of simulation: Syscall Emulation (SE) and Full System (FS). SE simulates only the CPU and the memory system. It translates system calls directly to the host system, allowing for faster simulation but without modeling a full operating system. FS emulates the entire hardware system and runs an unmodified kernel. This mode provides a complete simulated environment where an OS, along with all system components (such as memory, caches, and peripherals), is modeled. This thesis utilizes Full-System Simulation as it provides a more detailed performance counters report and it allows for a more accurate representation of RISC-V hardware behavior. Running in this mode requires a kernel and disk image to provide a functioning OS for the simulated system.

Building gem5 requires specifying the target architecture during compilation. For example, to build gem5 for RISC-V, one would use:

```
scons build/RISCV/gem5.opt -j$(nproc)
```

The valid ISAs are RISC-V, Arm, x86, Sparc, Power and Mips.

At the end of simulation, gem5 generates various outputs in the *m5out* folder which are crucial for performance analysis. Two key outputs that were used extensively are:

- **stats.txt**: This file contains detailed simulation CPU statistics such as instruction count and branch prediction accuracy and component specific features, such as register file reads/writes and cache accesses.
- **Checkpoints**: gem5 allows checkpointing at specific execution points by saving the architectural state of the simulated system. This enables resuming simulations from a particular state instead of restarting from the beginning, which is particularly useful for long-running experiments. Checkpoints can also be added in the source code of executables that run within the simulation. Lastly, gem5 allows resuming from checkpoints regardless of the CPU model used, meaning that even if the checkpoint was taken with one CPU model (e.g., O3CPU), the simulation can be resumed with a different CPU model (e.g., AtomicSimpleCPU), as long as the architectural state is compatible.

gem5 supports different CPU modes that balance speed and accuracy:

- **AtomicSimpleCPU**: The simplest in-order CPU mode in gem5, designed for speed over accuracy. It does not model detailed pipeline stages and timing, making it ideal for early-stage design exploration or when you need to simulate large-scale systems quickly.

- **TimingSimpleCPU**: This model simulates the timing of a simple CPU, providing more accuracy than AtomicSimpleCPU. It models the timing of basic operations like fetch, decode, execute, and memory access.
- **O3CPU (Out-of-Order CPU)**: This CPU model provides a more accurate representation of modern processors, including out-of-order execution, branch prediction, and other advanced features[16].
- **KvmCPU**: The KvmCPU model is used for simulating systems that run directly on a host machine using KVM (Kernel-based Virtual Machine).
- **MinorCPU**: This is a moderately detailed CPU model that balances performance and accuracy. It provides more pipeline stages and accuracy than the AtomicSimpleCPU and TimingSimpleCPU models but is less complex than O3CPU.

Finally, gem5 provides extensive configurability, allowing adjustments in various system parameters such as number of registers and cache memory configurations. This shaping can be defined in the gem5 configuration file, but some of the settings can be specified in the flags of the simulation initialization command as well.

3.2 Scripts

For the thesis, a checkpoint was added at the start of the chosen programs to perform isolated runs. These checkpoints were taken manually, by adding the checkpoint at the start of the main function in the source code, starting the simulation, copying and running the executable with the input file(s) (if any). Moreover, Python scripts were utilized to automate several tasks related to program checkpointing and execution analysis in gem5. One script was developed to automate the creation of program checkpoints using AtomicSimpleCPU by periodically capturing the architectural state after a specific number of instructions. Another script was used to resume execution from these newly created checkpoints using the O3CPU for a fixed number of cycles and saving the stats.txt files. Additionally, Jupyter Notebook files were created for regression analysis, using scikit-learn, pandas, numpy, and PRIM (Patient Rule Induction Method) from the Project-Platypus[2]. The regression methods used and the methodologies tried are explained in the next chapters.

4. REGRESSION AND CORRELATION ANALYSIS TECHNIQUES

In this chapter the necessary concepts and terminology used in the regression and correlation analysis are discussed.

4.1 Pearson's Coefficient

The Pearson correlation coefficient, denoted as r, is defined as:

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

where:

- x_i, y_i are the individual sample points,
- $\bar{x}$ and $\bar{y}$ are the means of the x and y values, respectively,
- n is the number of data points.

The Pearson correlation coefficient ranges from -1 to 1:

- $r = 1$: Perfect positive correlation (as x increases, y increases proportionally).
- $r > 0$: Positive correlation (as x increases, y tends to increase).
- $r = 0$: No correlation (no linear relationship between x and y).
- $r < 0$: Negative correlation (as x increases, y tends to decrease).
- $r = -1$: Perfect negative correlation (as x increases, y decreases proportionally).

4.2 Regression

The regression techniques used will be explained in detail in the following sections. In the context of this thesis, the process of applying these techniques based on a given parametric space will be referred to as the regression procedure.

4.2.1 Linear Regression

Linear regression is a statistical method used to model the relationship between a dependent variable y and one or more independent variables x. The simplest form, known as simple linear regression, is expressed as:

$$y = \beta_0 + \beta_1 x + \varepsilon$$

where:

- y is the dependent variable (response variable),
- x is the independent variable (predictor variable),
- β_0 is the intercept (the value of y when $x = 0$),
- β_1 is the slope of the regression line (indicating the change in y for a unit change in x),
- ε is the error term (representing the deviation of actual values from the predicted values).

When there are multiple independent variables, the model extends to:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n + \varepsilon$$

where $x_1, x_2, \ldots, x_n$ are the independent variables, and $\beta_1, \beta_2, \ldots, \beta_n$ are their corresponding coefficients.

4.2.2 Polynomial Regression

Polynomial regression is an extension of linear regression that models the relationship between the dependent variable y and the independent variable x as an n-th degree polynomial. The general form of polynomial regression is:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_n x^n + \varepsilon$$

where:

- y is the dependent variable,
- x is the independent variable,
- $\beta_0, \beta_1, \beta_2, \ldots, \beta_n$ are the regression coefficients,
- n is the degree of the polynomial,
- ε is the error term.

For multiple features, polynomial regression generates additional interaction terms. Consider two original features x_1 and x_2 transformed into polynomial features up to degree 2:

Original Features	Degree 1	Degree 2
x_1, x_2	x_1, x_2	$x_1^2, x_2^2, x_1 x_2$
x_3, x_4	x_3, x_4	$x_3^2, x_4^2, x_3 x_4$

For a polynomial of degree n, the number of generated features increases significantly due to interaction terms. For example, with three features (x_1, x_2, x_3), the second-degree polynomial features would be:

$$x_1, x_2, x_3, x_1^2, x_2^2, x_3^2, x_1x_2, x_1x_3, x_2x_3$$

These transformed features allow the model to capture nonlinear relationships between x and y, making polynomial regression a powerful technique for modeling complex patterns.

4.2.3 Ridge Normalization

Ridge regression is a type of linear regression that includes an L_2 regularization term to prevent overfitting by shrinking the model coefficients. The Ridge regression objective function is:

$$\min_{\beta_0,\beta} \sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{m} \beta_j x_{ij})^2 + \lambda \sum_{j=1}^{m} \beta_j^2$$

where:

- y_i is the dependent variable for the i-th observation,
- x_{ij} is the j-th feature for the i-th observation,
- β_0 is the intercept term,
- β_j are the regression coefficients,
- λ is the regularization parameter that controls the penalty on coefficients,
- n is the number of observations, and m is the number of features.

The regularization strength is controlled by λ:

- If $\lambda = 0$, Ridge behaves like an ordinary least squares regression.
- If λ is too large, the coefficients become very small, leading to an underfitted model.

4.2.4 Lasso Normalization

Lasso (Least Absolute Shrinkage and Selection Operator) is a type of regression that includes an L_1 regularization term to enforce sparsity in the model. The Lasso regression optimizes the following objective function:

$$\min_{\beta_0,\beta} \sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{m} \beta_j x_{ij})^2 + \lambda \sum_{j=1}^{m} |\beta_j|$$

where:

- y_i is the dependent variable for the i-th observation,
- x_{ij} is the j-th feature for the i-th observation,
- β_0 is the intercept term,
- β_j are the regression coefficients,
- λ is the regularization parameter that controls the penalty on coefficients,
- n is the number of observations, and m is the number of features.

The regularization strength is controlled by λ:

- If $\lambda = 0$, Lasso behaves like an ordinary least squares regression.
- If λ is too large, all coefficients shrink to zero, leading to an underfitted model.

Unlike Ridge regression, which applies an L_2-norm penalty ($\sum \beta_j^2$), Lasso can shrink some coefficients to zero, making it useful for feature selection.

4.2.5 Principal Component Analysis

Principal Component Analysis (PCA) is a dimensionality reduction technique that transforms a set of correlated features into a smaller set of uncorrelated features called principal components. It is commonly used in regression to reduce multicollinearity and improve model performance.

Given a dataset X with m features, PCA performs the following steps:

1. **Standardization**: Center and scale the data:

$$X_{\text{scaled}} = \frac{X - \mu}{\sigma}$$

   where μ and σ are the mean and standard deviation of each feature.

2. **Compute Covariance Matrix**:

$$C = \frac{1}{n} X_{\text{scaled}}^T X_{\text{scaled}}$$

3. **Eigen Decomposition**: Solve for eigenvalues and eigenvectors:

$$Cv = \lambda v$$

   The eigenvectors v form the principal components.

4. **Project Data onto Principal Components**:

$$Z = X_{\text{scaled}} V$$

   where V is the matrix of the top k eigenvectors corresponding to the largest eigenvalues.

After PCA is performed, the regression model is fit in the reduced space and then transformed back to obtain the original regression equation.

Given the transformation:

$$X_{\text{pca}} = X_{\text{scaled}} V$$

where X_{pca} is the transformed dataset, the regression model is trained as follows:

$$y = \beta_0 + \beta_1 Z_1 + \beta_2 Z_2 + \cdots + \beta_k Z_k$$

Since V is semi-orthogonal, to revert back to the original feature space :

$$X_{\text{orig}} = X_{\text{pca}} V^T$$

Thus, the regression equation in the original feature space is:

$$y = \beta_0 + (\beta_1 V_1^T + \beta_2 V_2^T + \cdots + \beta_k V_k^T) X_{\text{scaled}}$$

Finally, substituting X_{scaled} back in terms of the original features:

$$X_{\text{scaled}} = \frac{X - \mu}{\sigma}$$

The final regression equation is obtained:

$$y = \beta_0 + \sum_{j=1}^{m} \gamma_j x_j$$

where γ_j are the transformed regression coefficients.

Since PCA is designed primarily for dimensionality reduction rather than perfect reconstruction, these errors are expected. A higher number of retained components improves the reconstruction precision but increases model complexity.

4.3 Patient Rule Induction Method

The Patient Rule Induction Method (PRIM)[9] is a statistical technique used for identifying regions in the feature space where the response variable Y is particularly high (or low). Unlike traditional regression techniques, which focus on fitting a global model, PRIM is designed for discovering localized patterns in the data. It seeks to identify subregions (or ”boxes”) in the feature space where the target variable y takes extreme values, providing interpretable rules for decision-making.

Given a dataset with features $X = (x_1, x_2, \ldots, x_m)$ and a response variable Y, PRIM follows these steps:

1. **Initialization**: Consider the entire feature space as one large box containing all the data points.

2. **Peeling Stage**: Iteratively remove small fractions of the dataset (α_{peel}) (typically 5-10%) that do not contribute significantly to high values of Y, shrinking the box towards a high-response region. The criterion for removing a fraction of a feature value (either by reducing its upper bound or by increasing its upper bound) is based on maximizing/minimizing the chosen objective function in the remaining region. Two of the most popular objective functions are:

   - **Lenient1 Criterion:** The objective function evaluates the gain in mean divided by the loss in mass, as given by:

     $$\text{obj} = \frac{\text{ave}[Y_i \mid X_i \in B - b] - \text{ave}[Y \mid X \in B]}{|n(Y_i) - n(Y)|}$$

     where:
     - B is the original box,
     - $B - b$ is the set of candidate new boxes,
     - Y are the response values in the old box,
     - $n(Y_i)$ and $n(Y)$ are the cardinalities of Y_i and Y, respectively.
   - **Lenient2 Criterion:**

     $$\text{obj} = n(Y_i) \cdot \frac{\text{ave}[Y_i \mid X_i \in B - b] - \text{ave}[Y \mid X \in B]}{|n(Y) - n(Y_i)|}$$

     where B is the current subregion.

     The peeling stage terminates when the total number of points in the box (mass) is smaller than a given fraction or no contribution increases/decrease the value of the objective function.

3. **Pasting Stage**: After reaching a small region with high response, PRIM attempts to slightly expand the box back by iteratively adding small fractions of the dataset (α_{paste}). This step is similar to the second.

4. **Rule Extraction**: Once an optimal box is identified, PRIM extracts decision rules based on the ranges of the selected features defining the region.

The PRIM method can be evaluated based on these metrics:

- **Mean:** This is the average value of the response variable y for all data points in the box. It is defined as:

  $$\text{Mean} = \frac{1}{n_{\text{box}}} \sum_{i \in \text{box}} y_i,$$

  where n_{box} is the number of points in the box and y_i is 1 if it is a desirable data point, otherwise 0.

- **Mass:** This represents the proportion of the total data points that fall within the box. It is given by:

  $$\text{Mass} = \frac{n_{\text{box}}}{n_{\text{total}}},$$

  where n_{total} is the total number of points.

- **Coverage:** This metric measures the fraction of the total *desirable* points that are contained in the box. If $n_{\text{desirable, box}}$ is the number of desirable points in the box and $n_{\text{desirable, total}}$ is the total number of desirable points, then

$$\text{Coverage} = \frac{n_{\text{desirable, box}}}{n_{\text{desirable, total}}}.$$

- **Density:** This is the ratio of the number of desirable points in the box to the total number of points in the box. It is defined as:

$$\text{Density} = \frac{n_{\text{desirable, box}}}{n_{\text{box}}}.$$

This method is more preferable than choosing the hard limits of the features for a given threshold of the target value mainly because of these reasons:

- PRIM handles high-dimensional data and produces interpretable decision rules in terms of just a few feature boundaries.
- PRIM allows flexibility in defining peeling criteria, such as Lenient1 and Lenient2. The peeling criteria usually takes into consideration both the mean of the target value and the coverage and by doing so it reduces the impact of the noise data in the final box.

Finally, after the box is formed, each different parameter is assigned a quasi p-value[7]. In short, quasi p-value expresses the likelihood that a given parameter constraint could have occurred by chance. It takes values in the range $[0, 1]$, similar to a traditional p-value and has the following interpretation:

- 0: The parameter is highly significant, meaning the constraint is extremely unlikely to have occurred by random chance.
- **Close to** 0: The parameter plays an important role in distinguishing between scenarios.
- 0.5: The parameter constraint has a moderate likelihood of occurring by chance, meaning it may or may not be meaningful.
- 1: The parameter is completely uninformative, meaning its constraint is entirely likely to have occurred by random chance and does not contribute meaningfully to scenario discovery.

The parameters with quasi p-values equal to one are removed from the final boxes in this research.

5. IMPLEMENTATIONS OF AVF PREDICTION

5.1 Regression Analysis Setup

To predict the target values associated with the Architectural Vulnerability Factor (total-AVF, SDC-AVF, Timeout-AVF, Assertion/Crash-AVF) at runtime based on the initially selected features for every hardware component (Register File, L1 Data Cache and L1 Instruction Cache), the following steps are taken as the foundation for implementing regression analysis:

- **Step 1**: Divide the execution of the selected programs into intervals by taking checkpoints every 10 million instructions, while excluding the final incomplete interval.
- **Step 2**: Run each checkpoint for 10 million clock cycles in O3 CPU mode using the gem5 configuration shown in Table 5.1 and collect the feature counters for every hardware component.

Table 5.1: MAJOR SIMULATOR CONFIGURATIONS FOR EACH ISA (Adapted from [8])

Parameter	Value
ISA	RISC-V
Pipeline	64-bit OoO (8-issue)
L1 Instruction Cache	32KB, 64B line, 128 sets, 4-way
L1 Data Cache	32KB, 64B line, 128 sets, 4-way
L2 Cache	1MB, 64B line, 2048 sets, 8-way
Physical Register File	128 Int; 128 FP
LQ/SQ/IQ/ROB entries	32/32/64/128

- **Step 3**: Conduct 500 program executions in O3 CPU mode, starting the execution from the current checkpoint. In each execution, a single random bit flip is injected in the register or memory values of the specified hardware component within the first 10 million clock cycles. This was done utilizing the gem5-MARVEL infrastructure[8][19][18][13]. Based on the program's execution and output, errors are categorized as Masked, Silent Data Corruption (SDC), Timeout or Assertion/Crash. The Architectural Vulnerability Factor corresponding to the specific execution is calculated by adding the total SDCs, Timeouts and Assertions/Crashes.

  The number of executions, given the specific configuration, corresponds to the margin of errors[14] shown in 5.2:

Confidence	95%	99%	99.8%
Register File	4.4%	5.8%	6.9%
L1 Data Cache	4.4%	5.8%	7.0%
L1 Instruction Cache	4.4%	5.8%	7.0%

Table 5.2: Margin of error based on confidence levels

5.2 Formal Definition of the Regression Problem

The execution of a checkpoint formed in Step 2 defines a vector of counters for every component. Therefore, the vector that corresponds to the interval i can be represented as X_i. These will be the independent variables of the regression model.

The 500 batch of executions of a checkpoint formed in Step 3 defines a percentage value for every target value. Therefore, the target value that corresponds to the interval i can be represented as Y_i. This will be the dependent variable of the regression model.

Assuming that there are n checkpoints, the existing dataset is represented as:

$$\{(X_1, Y_1), (X_2, Y_2), ..., (X_n, Y_n)\}$$

where:

- X_i represents the independent variables (input features),
- Y_i represents the dependent variable (output),
- The task is to learn a function $f(X)$ that maps X to Y such that:

$$Y = f(X) + \epsilon$$

  where ϵ is an error term (noise).

To evaluate a regression model for the problem, after performing k-fold validation with k=5, the average R^2 score of all folds is chosen as the metric.

The initial features for each hardware component were chosen manually from the produced *stats.txt* file, based on their relevance to the component's functionality:

- **Register file**:
  1. system.cpu.ipc
  2. system.cpu.intRegfileReads
  3. system.cpu.intRegfileWrites
  4. system.cpu.rename.squashCycles
  5. system.cpu.rename.idleCycles
  6. system.cpu.rename.blockCycles
  7. system.cpu.rename.serializeStallCycles
  8. system.cpu.rename.runCycles
  9. system.cpu.rename.unblockCycles
  10. system.cpu.rename.renamedInsts
  11. system.cpu.rename.ROBFullEvents
  12. system.cpu.rename.IQFullEvents
  13. system.cpu.rename.LQFullEvents
  14. system.cpu.rename.SQFullEvents

15. system.cpu.rename.fullRegistersEvents
16. system.cpu.rename.renamedOperands
17. system.cpu.rename.intLookups
18. system.cpu.rename.committedMaps
19. system.cpu.rename.undoneMaps
20. system.cpu.rename.serializing
21. system.cpu.rename.tempSerializing
22. system.cpu.rename.skidInsts
23. system.cpu.rob.reads
24. system.cpu.rob.writes

- **L1 Data Cache**:

1. system.cpu.ipc
2. system.cpu.dcache.demandHits::cpu.data
3. system.cpu.dcache.demandMisses::cpu.data
4. system.cpu.dcache.demandAccesses::cpu.data
5. system.cpu.dcache.writebacks::writebacks
6. system.cpu.dcache.replacements
7. system.cpu.dcache.LoadLockedReq.hits::cpu.data
8. system.cpu.dcache.LoadLockedReq.misses::cpu.data
9. system.cpu.dcache.LoadLockedReq.accesses::cpu.data
10. system.cpu.dcache.ReadReq.hits::cpu.data
11. system.cpu.dcache.ReadReq.misses::cpu.data
12. system.cpu.dcache.ReadReq.accesses::cpu.data
13. system.cpu.dcache.StoreCondReq.hits::cpu.data
14. system.cpu.dcache.StoreCondReq.misses::cpu.data
15. system.cpu.dcache.StoreCondReq.accesses::cpu.data
16. system.cpu.dcache.SwapReq.hits::cpu.data
17. system.cpu.dcache.SwapReq.misses::cpu.data
18. system.cpu.dcache.SwapReq.accesses::cpu.data
19. system.cpu.dcache.WriteReq.hits::cpu.data
20. system.cpu.dcache.WriteReq.misses::cpu.data
21. system.cpu.dcache.WriteReq.accesses::cpu.data
22. system.cpu.dcache.tags.totalRefs
23. system.cpu.dcache.tags.dataAccesses

- **L1 Instruction Cache**:

1. system.cpu.ipc
2. system.cpu.icache.demandHits::cpu.inst

3. system.cpu.icache.demandMisses::cpu.inst
4. system.cpu.icache.demandAccesses::cpu.inst
5. system.cpu.icache.writebacks::writebacks
6. system.cpu.icache.writebacks::writebacks
7. system.cpu.icache.replacements
8. system.cpu.icache.ReadReq.hits::cpu.inst
9. system.cpu.icache.ReadReq.misses::cpu.inst
10. system.cpu.icache.ReadReq.accesses::cpu.inst
11. system.cpu.icache.tags.totalRefs
12. system.cpu.icache.tags.dataAccesses

Identifying these features, or a subset of them, as hardware proxies for each component is a central objective of this thesis.

5.3 Prediction Approaches

In this section the regression and prediction methods are presented and evaluated. The checkpoints were obtained from the execution of programs from the mibench suite[5]. They were gathered 364 checkpoints for the Register File and the L1 Data Cache and 203 checkpoints for the L1 Instruction Cache. After the k-fold validation is completed, two programs are used for testing purposes.

Below, the variations and the parametric space of the regression procedure are defined. These variations and parametric space will be applied in all the combinations of features which will be selected after feature selection. Testing the parametric space for every combination of features and every hardware component is extremely computationally intensive. For example, for the Register File there are 2^{24} possible combinations of features. Therefore, other feature selections approaches are tried instead of brute force.

The key concepts behind the feature selection approaches in 5.3.1 and 5.3.2 are described here[21]. However, this thesis differs in several aspects, including the architecture (RISC-V instead of PISA), interval size (10 million clock cycles instead of 4 million instructions), CPU configuration and targeted hardware components. Additionally, fewer features are selected for each component compared to the 160 features used in the referenced study.

The small number of features for every hardware component allows for the exploration of other feature selection approaches, such as the approach in 5.3.3.

The PRIM algorithm is applied not only to the initial features of each hardware component but also to the features selected from the best model of the regression procedure. A similar approach is explored in [15], but with a different regression approach.

Finally, the same methodologies can be applied to other hardware components by following the exact same steps, with the only difference being the initial feature selection.

5.3.1 Linear Procedure

This approach is based on the linear procedure found in 4.2.1 here[21].

1. Choose the feature with the highest Pearson's correlation.
2. For each iteration, choose the next feature based on the best evaluation (R^2 score) of the regression procedure.
3. Repeat step 2 until the desired number of maximum features is reached.

5.3.2 Quadratic Procedure

This approach is based on the quadratic procedure found in 4.2.2 here[21].

1. Gather all possible pairs of features (x, y).
2. Perform second degree regression procedure for every pair using the basis $\{1, x, y, xy, x^2, y^2\}$.

5.3.3 Polynomial Pearson's Coefficient

This approach presents an alternative feature selection method that considers the Pearson's correlation of the polynomial features.

1. Gather the Pearson's correlations of the polynomial features for a given degree.
2. Choose the desired number of features that's members of the most correlated polynomial features and perform the regression procedure.
3. Repeat step 3 until all the desired numbers of features are checked.
4. Go to step 1 until all the desired degrees are checked.

5.4 Parametric Space

The parameters that define the parametric space are as follows:

- **Number of features**: The total number of features used in the regression. It is defined as $\{1, 2, ..., number\_of\_total\_features\}$, however the best solution for the set $\{1, 2, 3, 4, 5\}$ is also examined.
- **Degree**: The degree of the polynomial regression. It is fixed for linear and quadratic procedures (1 and 2 respectively), while for the Pearson correlation-based approach it is $\{1, 2, 3, 4\}$. For a feature space of $\{1, 2, 3, 4, 5\}$ the degree is limited to $\{1, 2, 3\}$.
- **Alpha**: The alpha parameter in ridge an lasso regression. The range of examined values for alpha is $[0.0001, 0.001, 0.01, 0.1, 1, 10, 100]$ for both types of regression.

The regression models evaluated in this thesis include the default, lasso and ridge regression, both in linear and polynomial forms.

6. RESULTS

In this chapter we present the final results for the total-AVF and SDC-AVF runtime predictions across all three approaches and components. Timeout-AVF and Assertion/Crash-AVF were also tested, but their results were insignificant, as not even the slightest correlation could be observed between the measured features and these failure outcomes. For each case, the results of the approach with the best R^2 score is presented.

6.1 All Features Included

6.1.1 total-AVF

6.1.1.1 Register File

Method: Linear Procedure **Model**: Ridge **Alpha**: 0.001 **Degree**: 1 **PCA components**: 10 R^2 **Score**: 78%

Formula:

$$
\begin{aligned}
&0.01756 \cdot system.cpu.rename.idleCycles \\
&+ 0.01040 \cdot system.cpu.rename.renamedOperands \\
&- 0.00760 \cdot system.cpu.rename.renamedInsts \\
&- 0.00469 \cdot system.cpu.rename.serializing \\
&- 0.00990 \cdot system.cpu.rename.blockCycles \\
&- 0.01026 \cdot system.cpu.rename.tempSerializing \\
&- 0.02536 \cdot system.cpu.ipc \\
&- 0.02624 \cdot system.cpu.rename.fullRegistersEvents \\
&- 0.02597 \cdot system.cpu.rename.intLookups \\
&- 0.02622 \cdot system.cpu.rename.runCycles \\
&+ 0.09085
\end{aligned}
$$

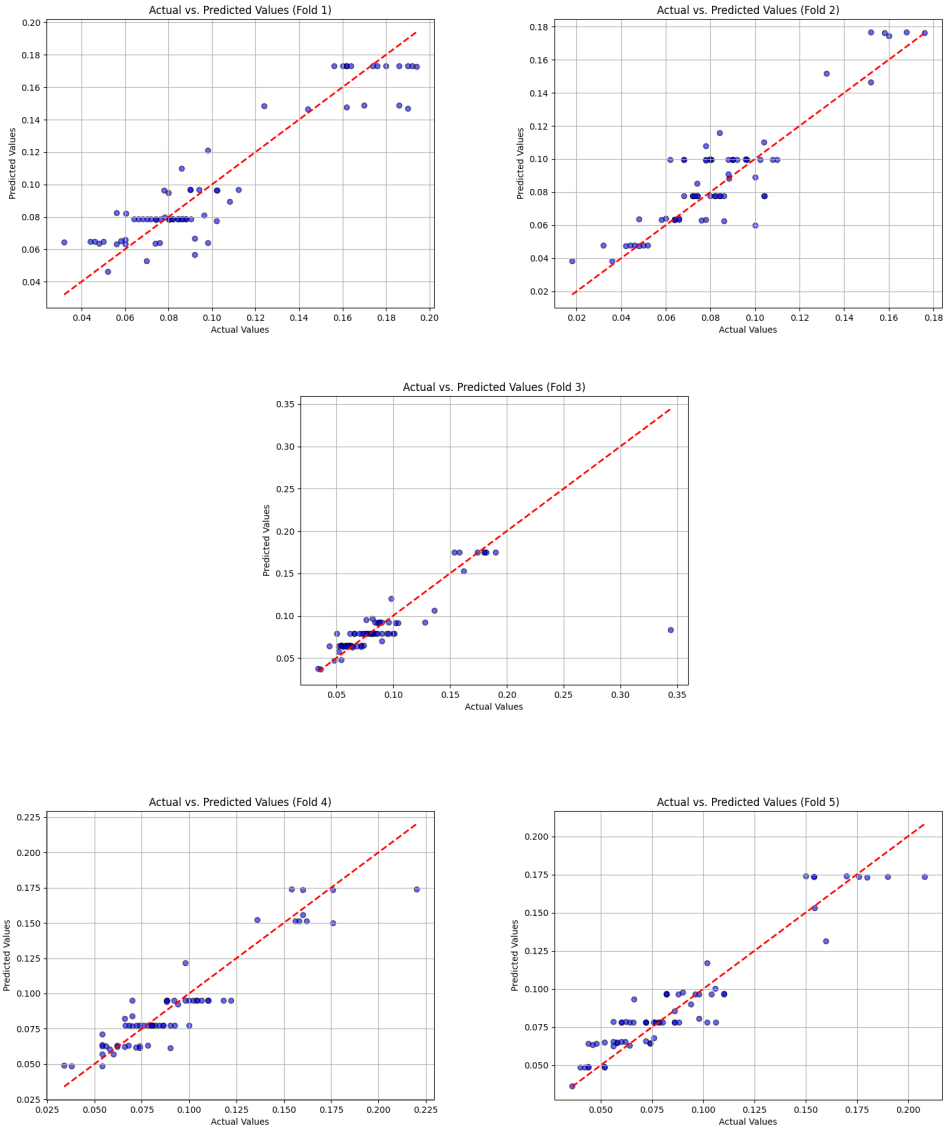

Figure 6.1: total-AVF, Register File, 24 maximum features, K-fold validation

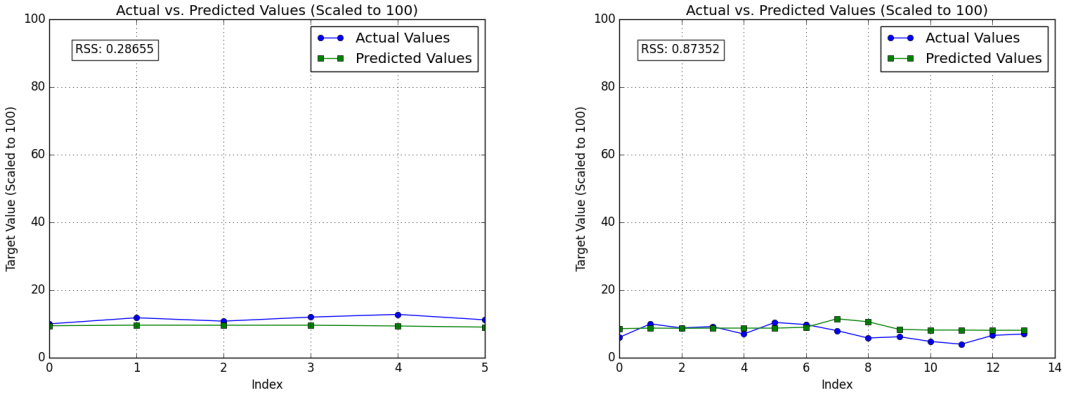

Figure 6.2: total-AVF, Register File, 24 maximum features, Test Programs

PRIM

Threshold: 10%

Table 6.1: total-AVF, Register File, 24 maximum features, PRIM statistics

Metric	Value
Coverage	0.647059
Density	1
Mass	0.160350
Mean	1

Table 6.2: total-AVF, Register File, 24 maximum features, PRIM box

Metric	Min	Max	QP Values
system.cpu.rename.renamedOperands	24309020	45842460	$1.40e-01$
system.cpu.rename.fullRegistersEvents	7567	924236	$3.71e-01$

6.1.1.2 L1 Data Cache

Method: Polynomial Pearson's Correlation **Model**: Ridge **Alpha**: 100 **Degree**: 4 **PCA components**: 12 R^2 **Score**: 87.9%

Formula: Too large

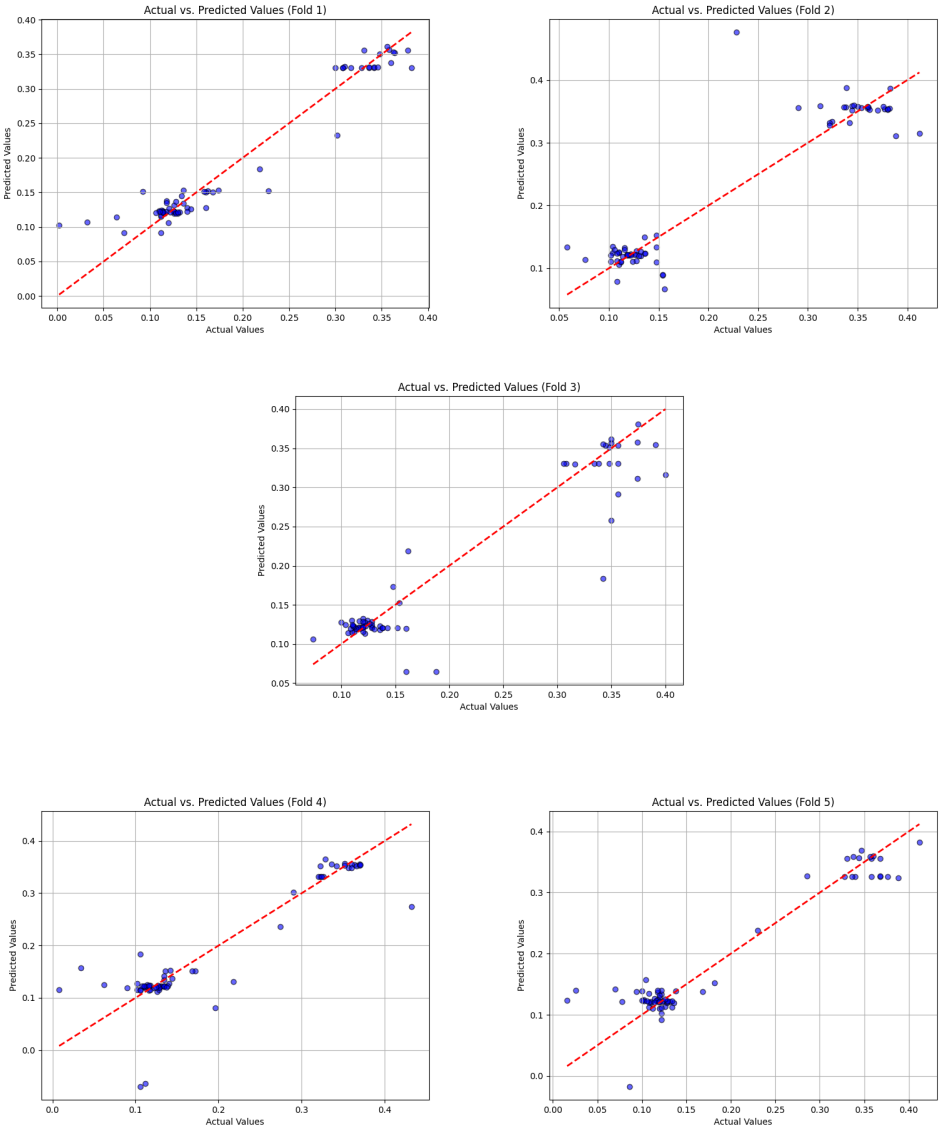

Figure 6.3: total-AVF, Data Cache, 23 maximum features, K-fold validation

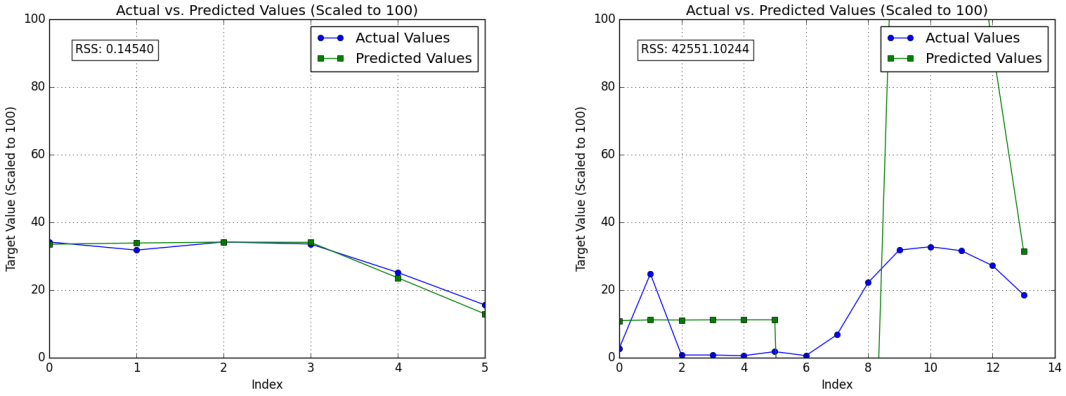

Figure 6.4: total-AVF, Data Cache, 23 maximum features, Test Programs

PRIM

Threshold: 30%

Table 6.3: total-AVF, Data Cache, 23 maximum features, PRIM statistics

Metric	Value
Coverage	0.970874
Density	0.952381
Mass	0.306122
Mean	0.952381

Table 6.4: total-AVF, Data Cache, 23 maximum features, PRIM box

Metric	Min	Max	QP Values
system.cpu.dcache.demandMisses::cpu.data	1292	62164	$2.54e-16$
system.cpu.ipc	2.1	4.3	$2.89e-09$
system.cpu.dcache.tags.totalRefs	3700976	12928680	$2.59e-02$
system.cpu.dcache.WriteReq.accesses::cpu.data	5418	3188463	$6.16e-01$

6.1.1.3 L1 Instruction Cache

Method: Polynomial Pearson's Correlation **Model**: Lasso **Alpha**: 0.0001 **Degree**: 3 **PCA components**: 11 R^2 **Score**: 90%

Formula: Too large

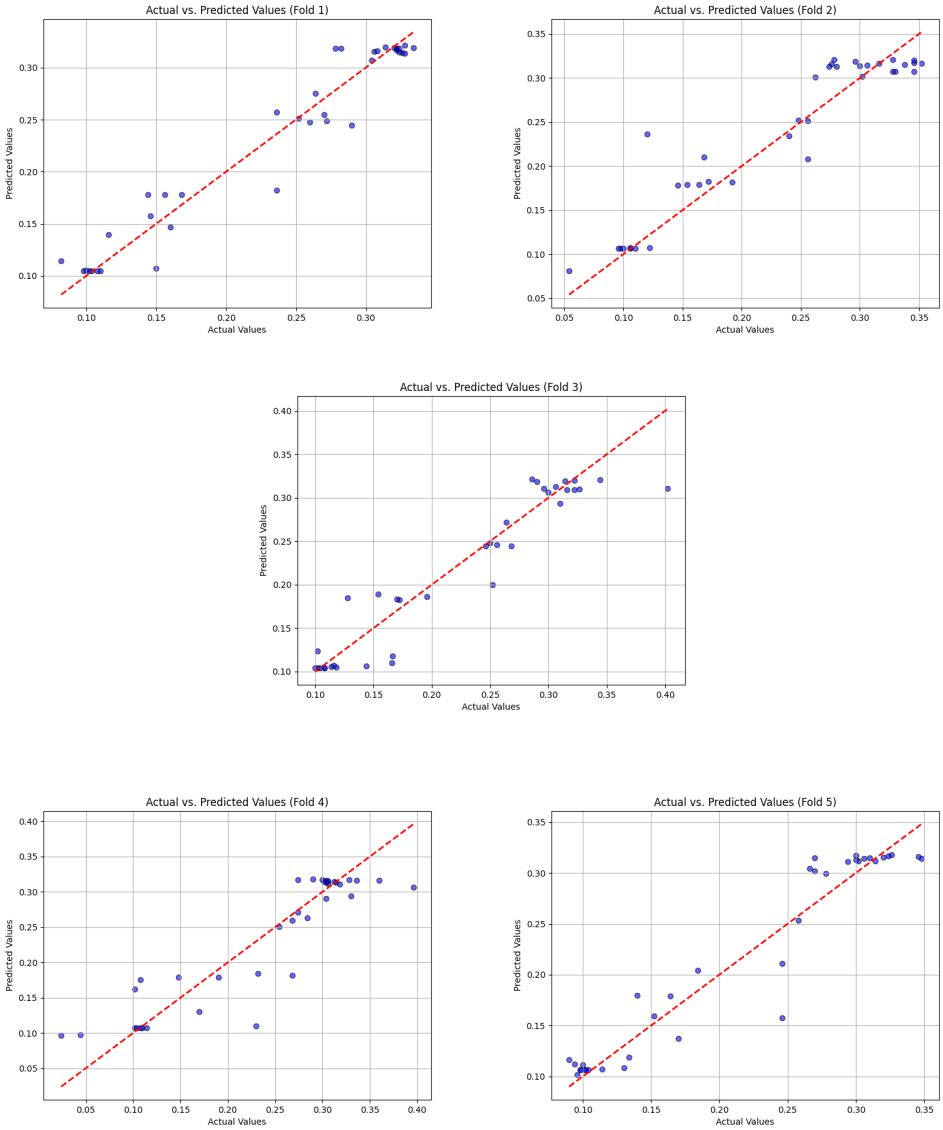

Figure 6.5: total-AVF, Instruction Cache, 12 maximum features, K-fold validation

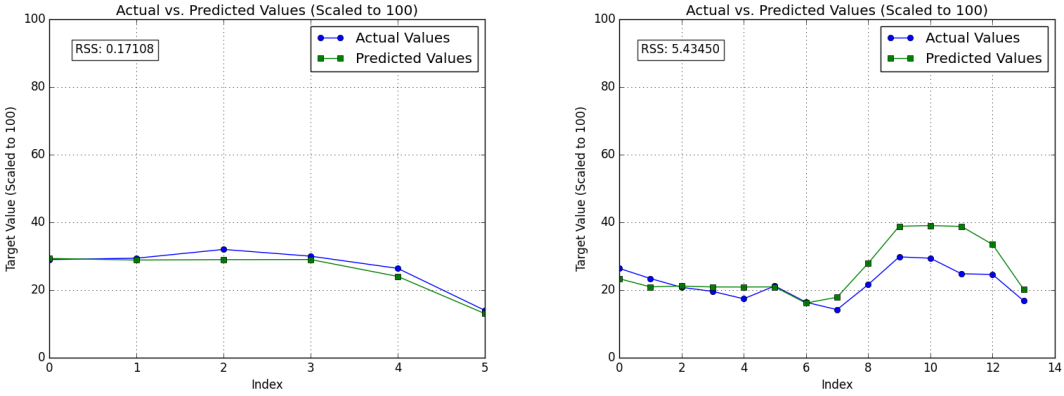

Figure 6.6: total-AVF, Instruction Cache, 12 maximum features, Test Programs

PRIM

Threshold: 30%

Table 6.5: total-AVF, Instruction Cache, 12 maximum features, PRIM statistics

Metric	Value
Coverage	0.982143
Density	0.753425
Mass	0.401099
Mean	0.753425

Table 6.6: total-AVF, Instruction Cache, 12 maximum features, PRIM box

Metric	Min	Max	QP Values
system.cpu.icache.writebacks::writebacks	12166	158529	$5.25e-04$
system.cpu.ipc	1.2	2.0	$1.53e-01$
system.cpu.icache.demandAccesses::cpu.inst	2119102	3314339	$5.63e-01$

6.1.2 SDC-AVF

6.1.2.1 Register File

Method: Linear Procedure **Model**: Ridge **Alpha**: 0.01 **Degree**: 1 **PCA components**: 13 R^2 **Score**: 87.5% **Formula**:

$$
\begin{aligned}
&0.02016 \cdot system.cpu.rename.renamedOperands \\
&- 0.00568 \cdot system.cpu.intRegfileWrites \\
&- 0.01215 \cdot system.cpu.rename.fullRegistersEvents \\
&- 0.02313 \cdot system.cpu.rename.squashCycles \\
&- 0.02115 \cdot system.cpu.rename.renamedInsts \\
&- 0.02426 \cdot system.cpu.rename.serializing \\
&- 0.02014 \cdot system.cpu.rename.IQFullEvents \\
&- 0.01980 \cdot system.cpu.rename.tempSerializing \\
&- 0.01970 \cdot system.cpu.rename.intLookups \\
&- 0.01976 \cdot system.cpu.rename.runCycles \\
&- 0.01969 \cdot system.cpu.rename.SQFullEvents \\
&- 0.01960 \cdot system.cpu.rename.idleCycles \\
&- 0.01953 \cdot system.cpu.rename.serializeStallCycles \\
&+ 0.03480
\end{aligned}
$$

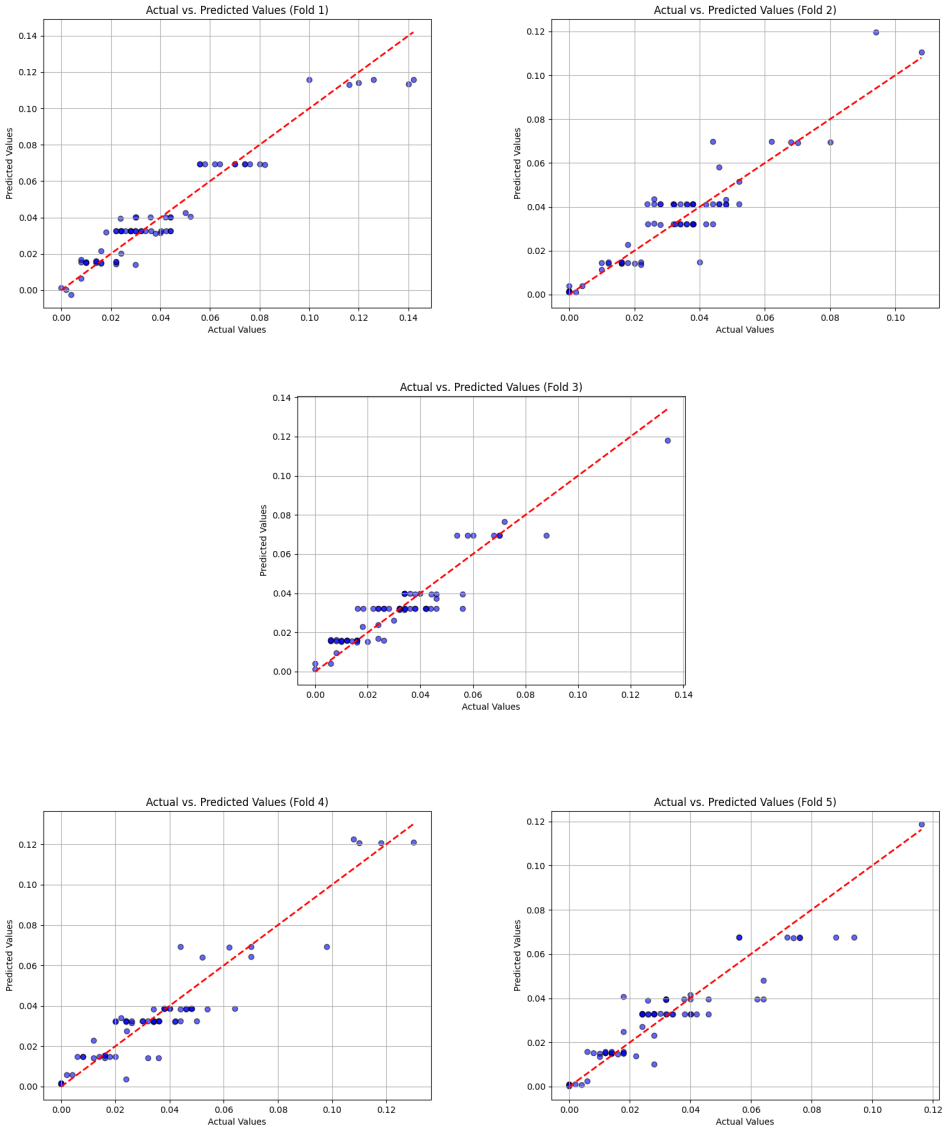

Figure 6.7: SDC-AVF, Register File, 24 maximum features, K-fold validation

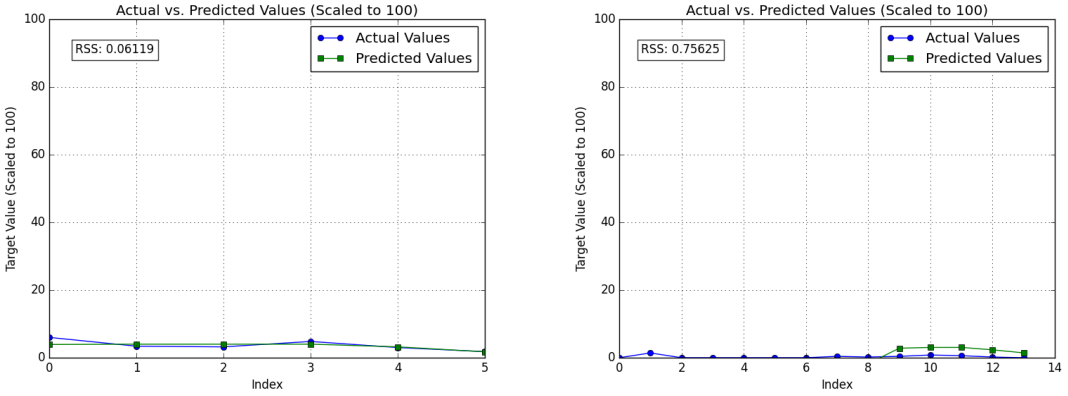

Figure 6.8: SDC-AVF, Register File, 24 maximum features, Test Programs

PRIM

Threshold: 8%

Table 6.7: SDC-AVF, Register File, 24 maximum features, PRIM statistics

Metric	Value
Coverage	0.789474
Density	1
Mass	0.043732
Mean	1

Table 6.8: SDC-AVF, Register File, 24 maximum features, PRIM box

Metric	Min	Max	QP Values
system.cpu.rename.SQFullEvents	76	12641	$2.88e-02$
system.cpu.rename.renamedOperands	26037042	45826733	$6.49e-02$
system.cpu.rename.fullRegistersEvents	7529	924236	$3.80e-01$
system.cpu.rename.IQFullEvents	15627	2167188	$3.80e-01$

6.1.2.2 L1 Data Cache

Method: Polynomial Pearson's Correlation **Model**: Ridge **Alpha**: 100 **Degree**: 4 **PCA components**: 10 R^2 **Score**: 90.8%

Formula: Too large

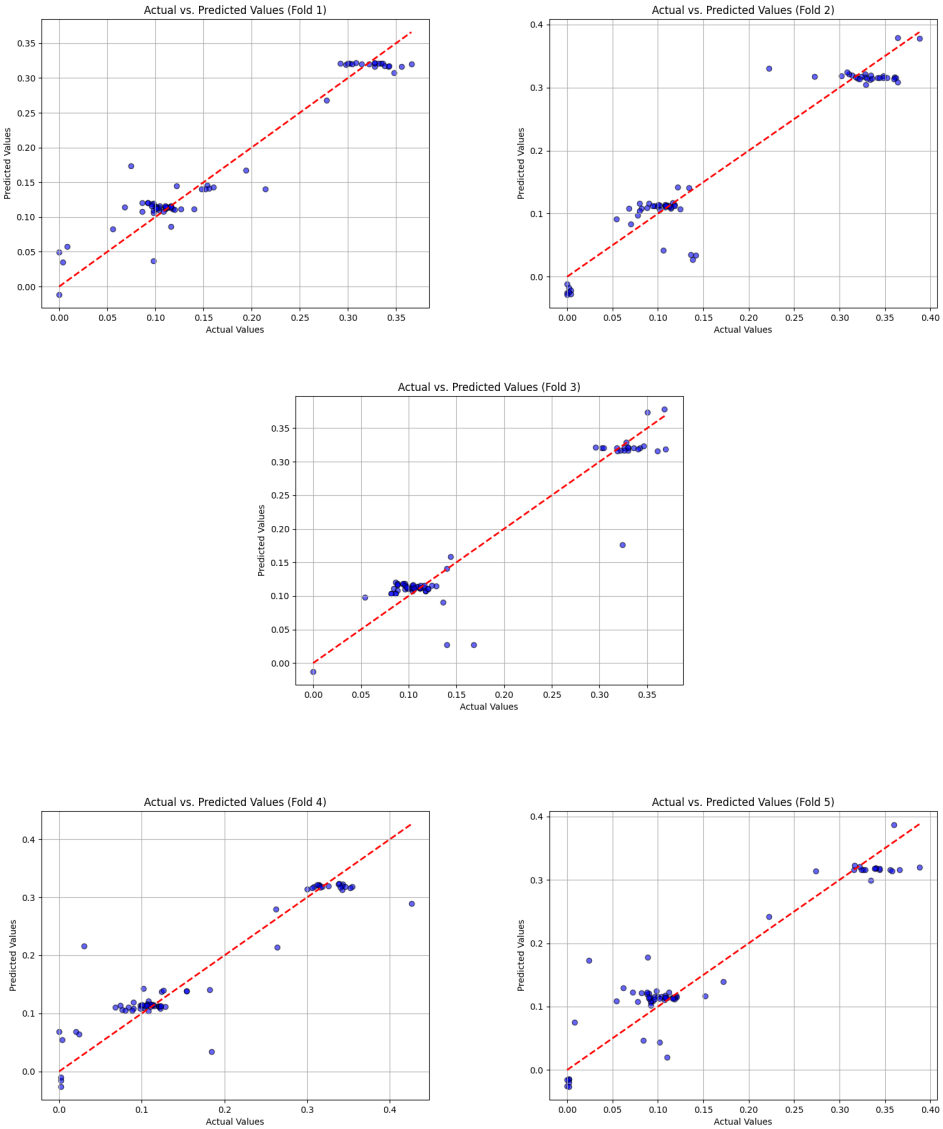

Figure 6.9: SDC-AVF, Data Cache, 23 maximum features, K-fold validation

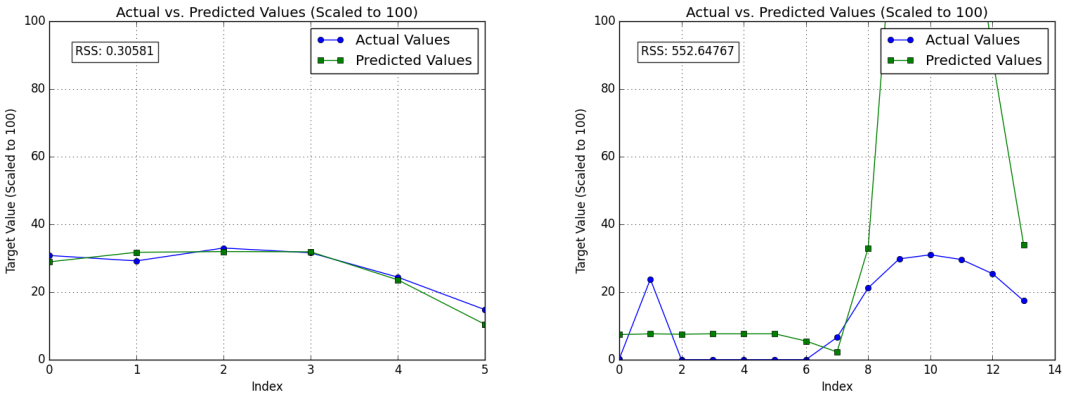

Figure 6.10: SDC-AVF, Data Cache, 23 maximum features, Test Programs

PRIM

Threshold: 30%

Table 6.9: SDC-AVF, Data Cache, 23 maximum features, PRIM statistics

Metric	Value
Coverage	0.959184
Density	0.940000
Mass	0.291545
Mean	0.940000

Table 6.10: SDC-AVF, Data Cache, 23 maximum features, PRIM box

Metric	Min	Max	QP Values
system.cpu.dcache.ReadReq.misses::cpu.data	719	35060	$4.07e-21$
system.cpu.dcache.demandHits::cpu.data	3965947	10751620	$3.33e-01$
system.cpu.ipc	2.1	4.3	$3.33e-01$
system.cpu.dcache.tags.dataAccesses	20769730	52191840	$3.33e-01$
system.cpu.dcache.WriteReq.hits::cpu.data	4606	3190222	$6.06e-01$

6.1.2.3 L1 Instruction Cache

Method: Polynomial Pearson's Correlation **Model**: Ridge **Alpha**: 10 **Degree**: 2 **PCA components**: 6 R^2 **Score**: 76.8%

Formula: Too large

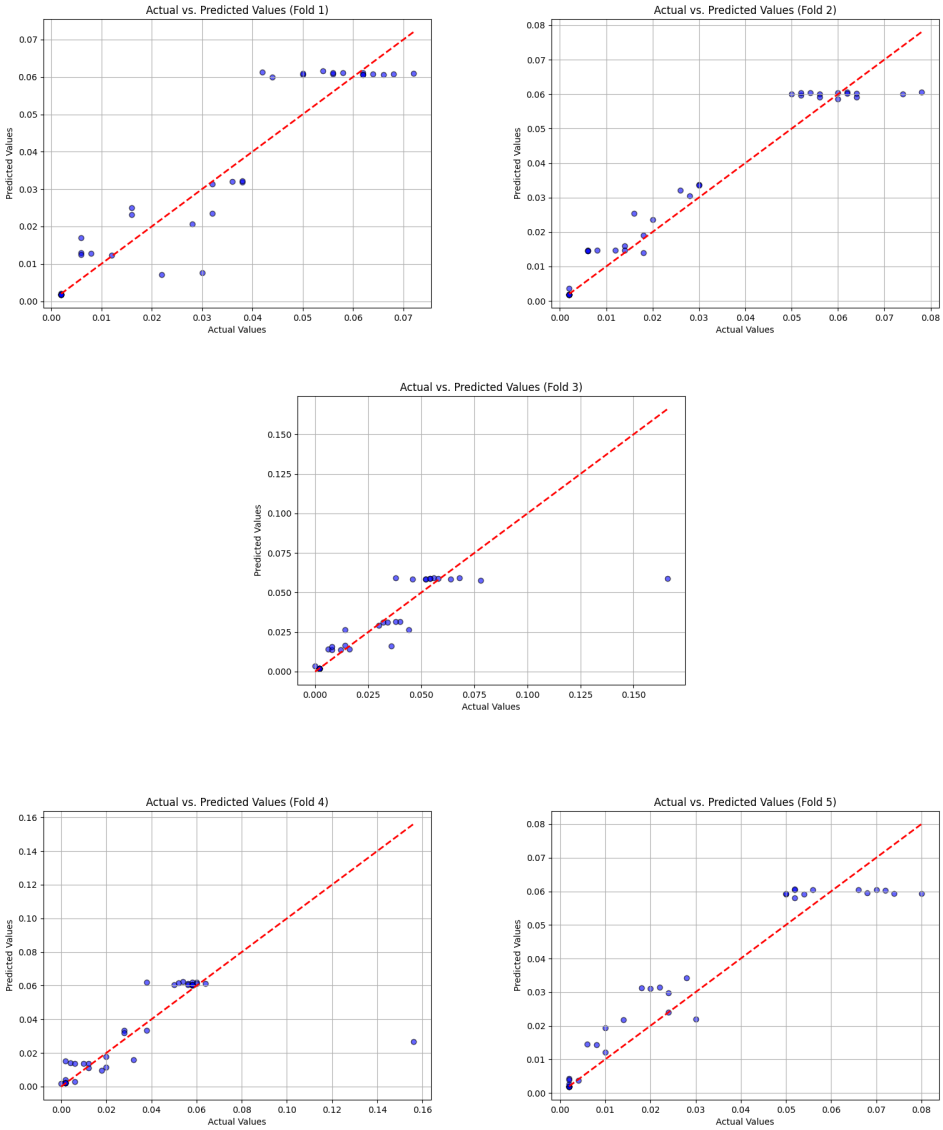

Figure 6.11: SDC-AVF, Instruction Cache, 12 maximum features, K-fold validation

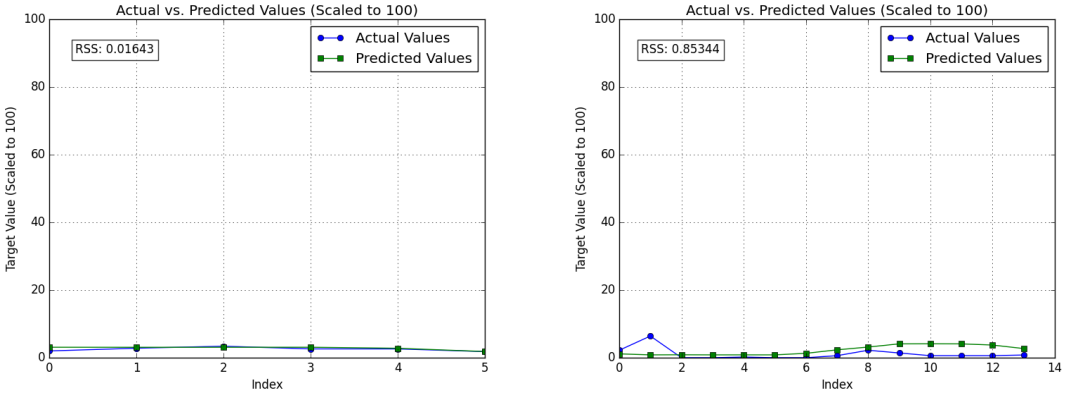

Figure 6.12: SDC-AVF, Instruction Cache, 12 maximum features, Test Programs

PRIM

Threshold: 10%

Table 6.11: SDC-AVF, Instruction Cache, 12 maximum features, PRIM statistics

Metric	Value
Coverage	1
Density	1
Mass	0.010989
Mean	1

Table 6.12: SDC-AVF, Instruction Cache, 12 maximum features, PRIM box

Metric	Min	Max	QP Values
system.cpu.icache.tags.dataAccesses	163627	9594045	$2.50e-01$
system.cpu.icache.demandMisses::cpu.inst	2197255	3300454	$4.44e-01$

6.2 Max of 5 Features Included

6.2.1 total-AVF

6.2.1.1 Register File

Method: Linear Procedure **Model**: Ridge **Alpha**: 0.1 **Degree**: 1 **PCA components**: 5 R^2 **Score**: 76.2%

Formula:

$$
\begin{aligned}
&0.02706 \cdot system.cpu.rename.idleCycles \\
&+ 0.02018 \cdot system.cpu.rename.renamedOperands \\
&- 0.00692 \cdot system.cpu.rename.renamedInsts \\
&- 0.00379 \cdot system.cpu.rename.serializing \\
&- 0.00733 \cdot system.cpu.rename.blockCycles \\
&+ 0.09085
\end{aligned}
$$

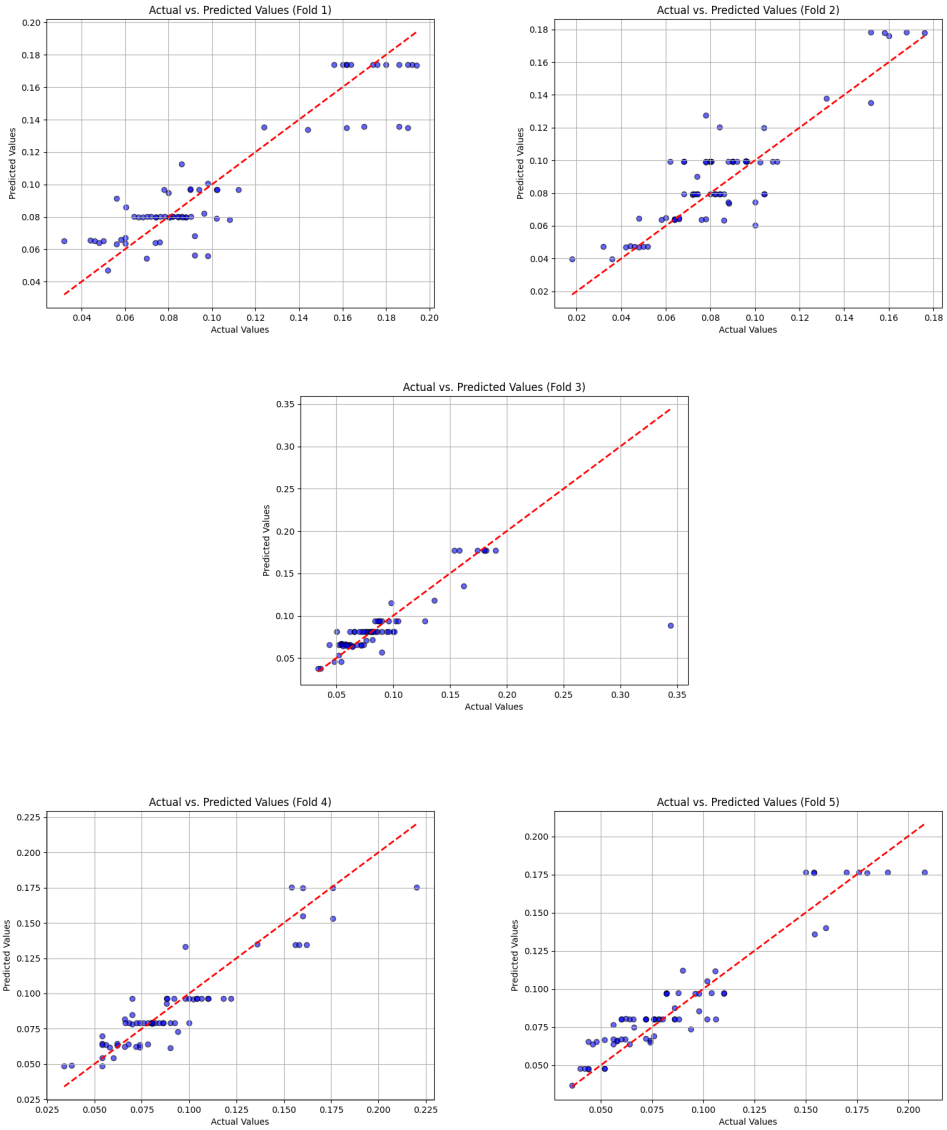

Figure 6.13: total-AVF, Register File, 5 maximum features, K-fold validation

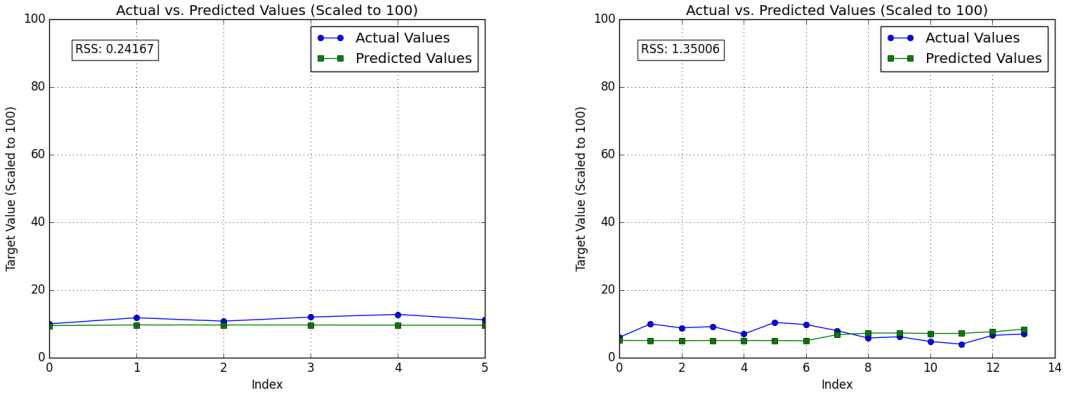

Figure 6.14: total-AVF, Register File, 5 maximum features, Test Programs

PRIM

Threshold: 10%

Table 6.13: total-AVF, Register File, 5 maximum features, PRIM statistics

Metric	Value
Coverage	0.647059
Density	0.982143
Mass	0.163265
Mean	0.982143

Table 6.14: total-AVF, Register File, 5 maximum features, PRIM box

Metric	Min	Max	QP Values
system.cpu.rename.blockCycles	207688	1111008	$2.10e-02$
system.cpu.rename.idleCycles	1134519	3122663	$2.55e-02$
system.cpu.rename.renamedInsts	22832425	50039387	$1.40e-01$
system.cpu.rename.renamedOperands	18053002	45842465	$3.71e-01$
system.cpu.rename.serializing	532	64018	$3.71e-01$

6.2.1.2 L1 Data Cache

Method: Polynomial Pearson's Correlation **Model**: Lasso **Alpha**: 0.01 **Degree**: 3 **PCA components**: 5 R^2 **Score**: 61.8%

Formula: Too large

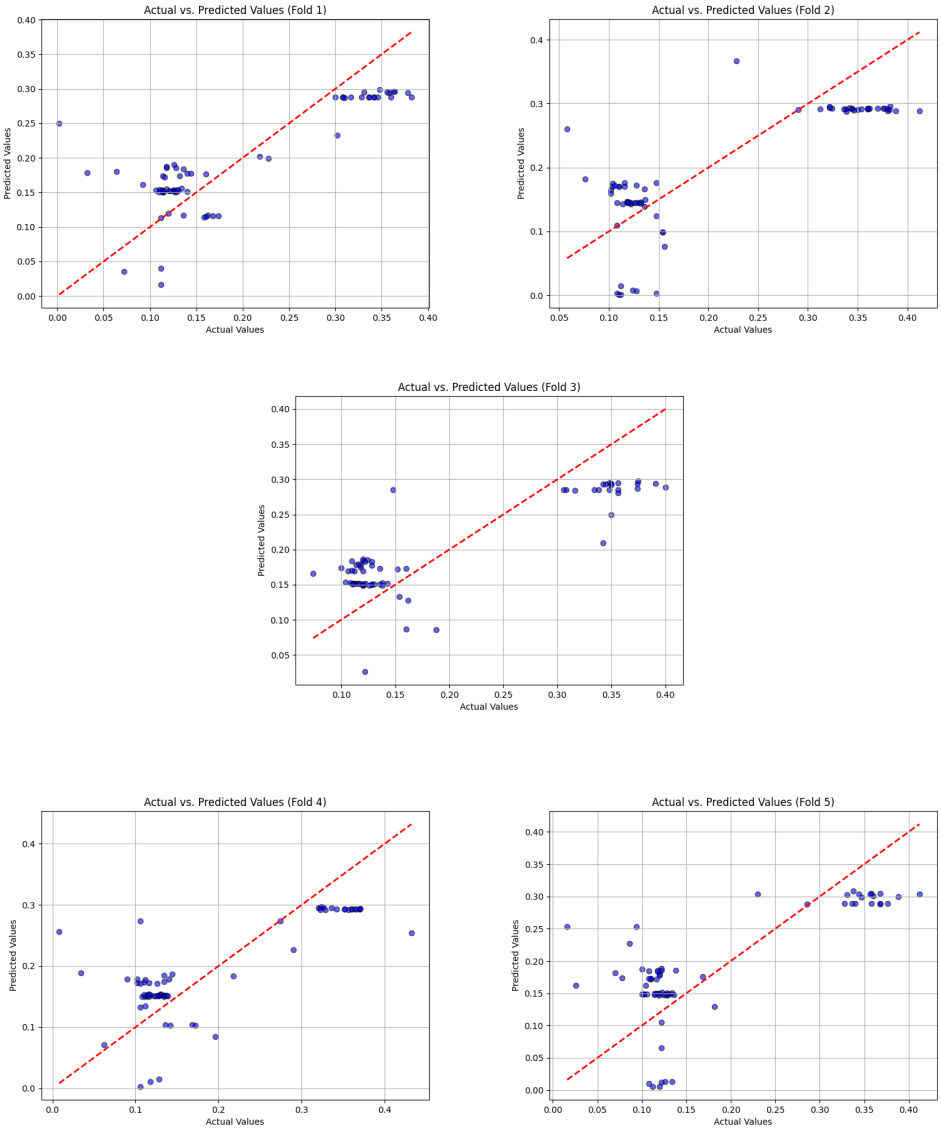

Figure 6.15: total-AVF, Data Cache, 5 maximum features, K-fold validation

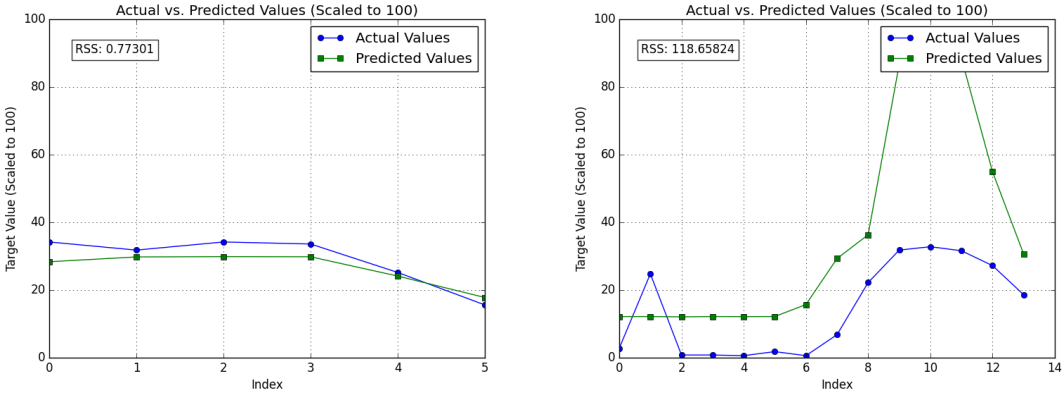

Figure 6.16: total-AVF, Data Cache, 5 maximum features, Test Programs

PRIM

Threshold: 30%

Table 6.15: total-AVF, Data Cache, 5 maximum features, PRIM statistics

Metric	Value
Coverage	0.970874
Density	0.943396
Mass	0.309038
Mean	0.943396

Table 6.16: total-AVF, Data Cache, 5 maximum features, PRIM box

Metric	Min	Max	QP Values
system.cpu.dcache.ReadReq.misses::cpu.data	719	34631	$2.95e-27$
system.cpu.ipc	2.1	4.3	$7.63e-03$
system.cpu.dcache.WriteReq.accesses::cpu.data	20480	3192842	$1.42e-01$
system.cpu.dcache.WriteReq.misses::cpu.data	362	35496	$6.06e-01$

6.2.1.3 L1 Instruction Cache

Method: Polynomial Pearson's Correlation **Model**: Ridge **Alpha**: 0.1 **Degree**: 2 **PCA components**: 4 R^2 **Score**: 87.8%

Formula: Too large

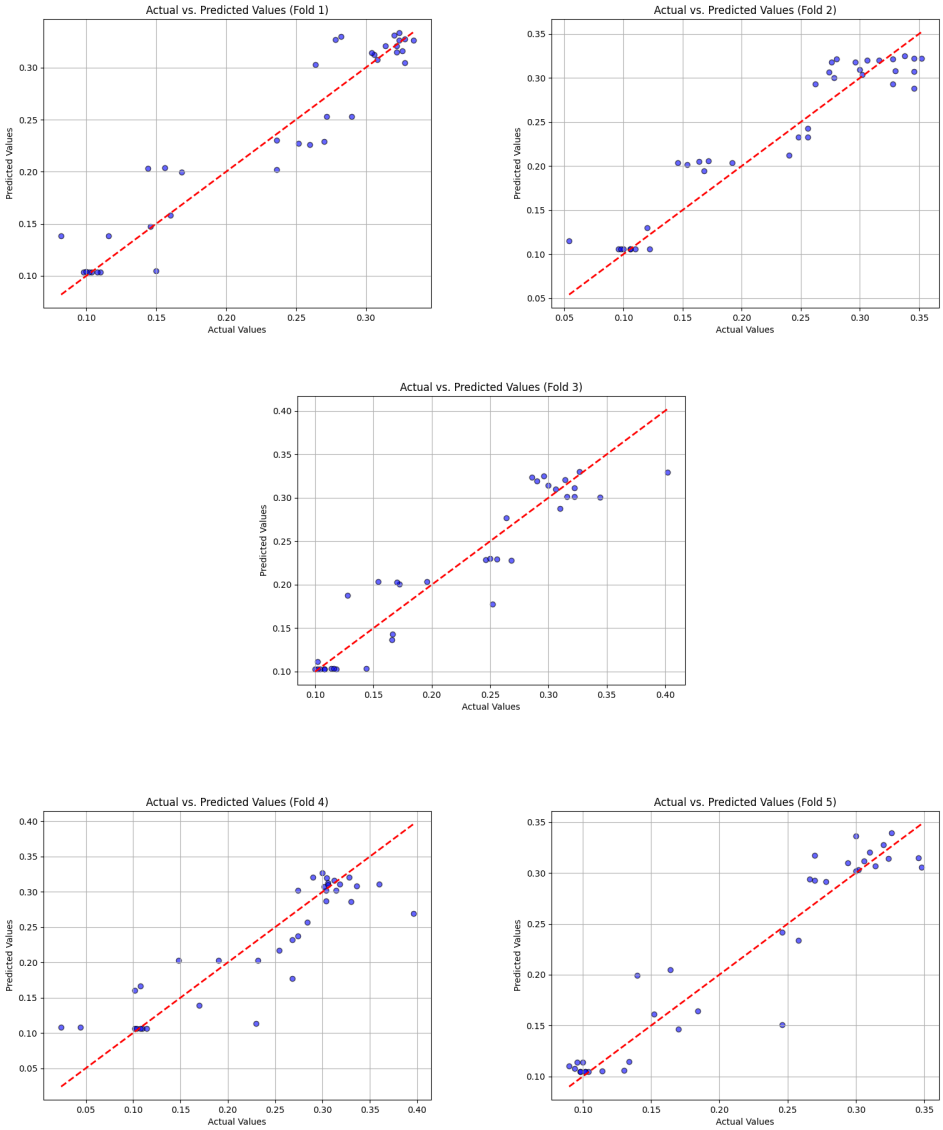

Figure 6.17: total-AVF, Instruction Cache, 5 maximum features, K-fold validation

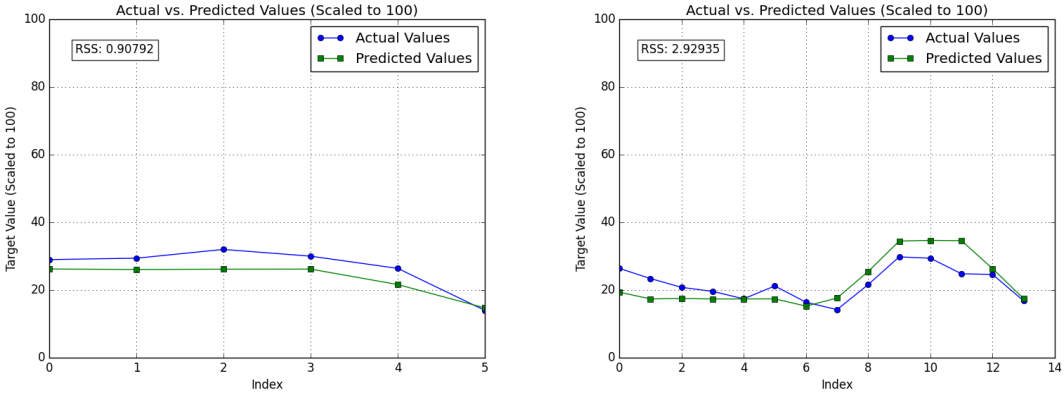

Figure 6.18: total-AVF, Instruction Cache, 5 maximum features, Test Programs

PRIM

Threshold: 30%

Table 6.17: total-AVF, Instruction Cache, 5 maximum features, PRIM statistics

Metric	Value
Coverage	0.857143
Density	0.8
Mass	0.329670
Mean	0.8

Table 6.18: total-AVF, Instruction Cache, 5 maximum features, PRIM box

Metric	Min	Max	QP Values
system.cpu.icache.ReadReq.misses::cpu.inst	163037	172223	$4.09e-01$
system.cpu.icache.replacements	149213	158529	$4.76e-01$
system.cpu.icache.tags.dataAccesses	9415305	9655692	$5.02e-01$

6.2.2 SDC-AVF

6.2.2.1 Register File

Method: Linear Procedure **Model**: Ridge **Alpha**: 0.01 **Degree**: 1 **PCA components**: 5 R^2 **Score**: 84.3%

Formula:

$$
\begin{aligned}
&0.03689 \cdot system.cpu.rename.renamedOperands \\
&+ 0.01109 \cdot system.cpu.intRegfileWrites \\
&+ 0.00130 \cdot system.cpu.rename.fullRegistersEvents \\
&- 0.01493 \cdot system.cpu.rename.squashCycles \\
&- 0.02462 \cdot system.cpu.rename.renamedInsts \\
&+ 0.03480
\end{aligned}
$$

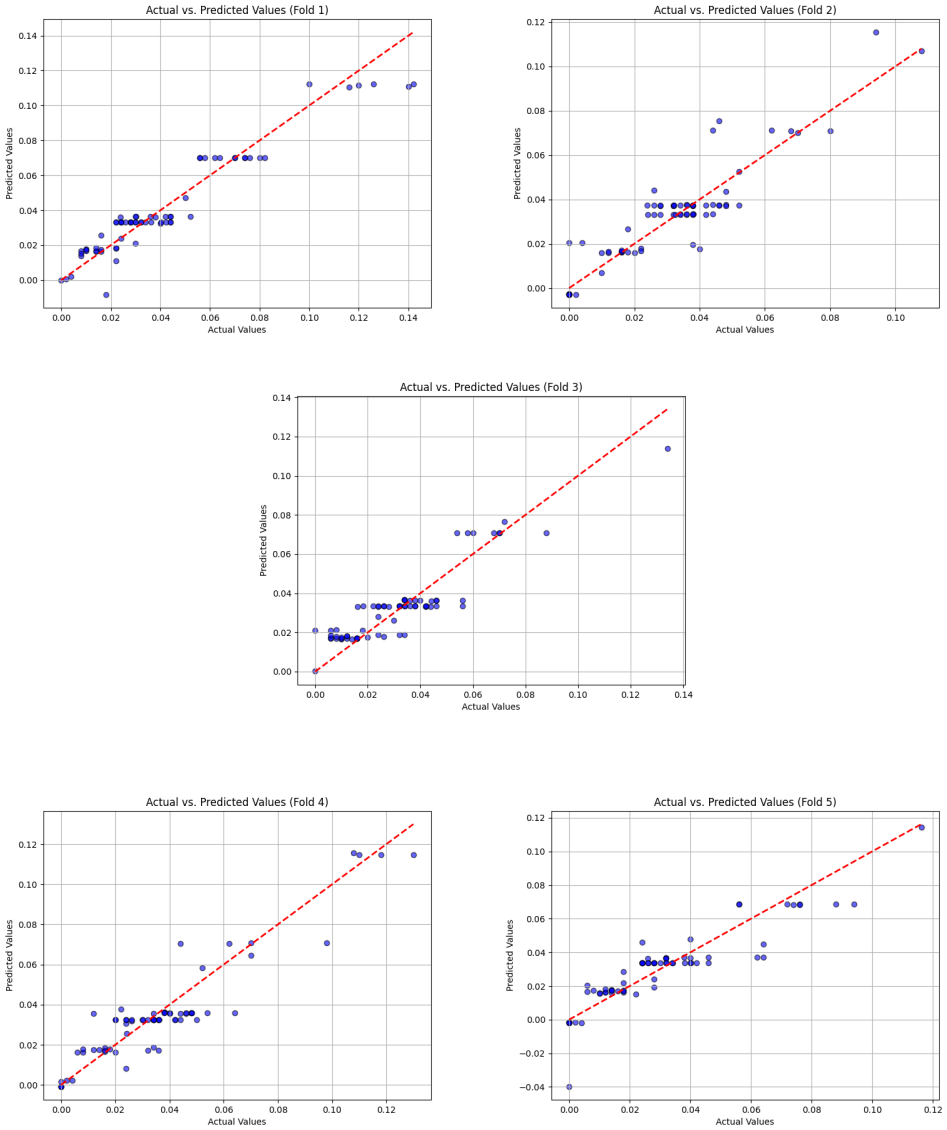

Figure 6.19: SDC-AVF, Register File, 5 maximum features, K-fold validation

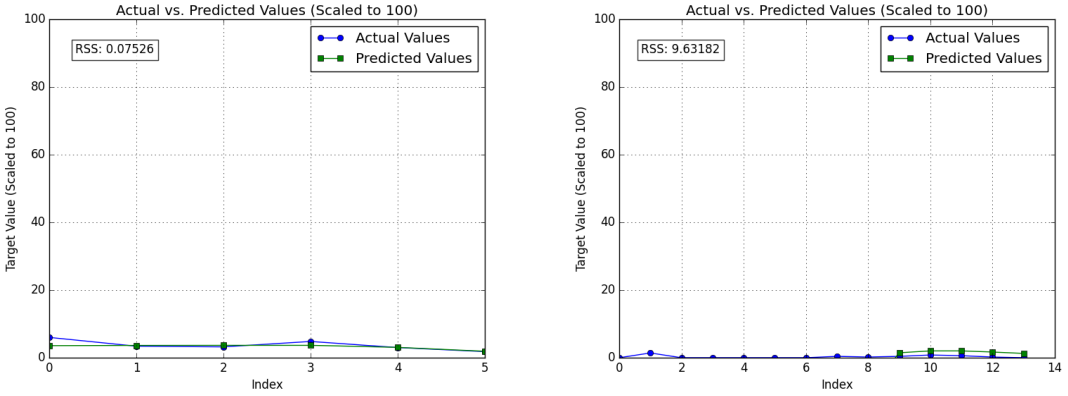

Figure 6.20: SDC-AVF, Register File, 5 maximum features, Test Programs

PRIM

Threshold: 8%

Table 6.19: SDC-AVF, Register File, 5 maximum features, PRIM statistics

Metric	Value
Coverage	0.736842
Density	1
Mass	0.040816
Mean	1

Table 6.20: SDC-AVF, Register File, 5 maximum features, PRIM box

Metric	Min	Max	QP Values
system.cpu.rename.fullRegistersEvents	694018	843135	$0.00e+00$
system.cpu.rename.squashCycles	3233	62035	$0.00e+00$

6.2.2.2 L1 Data Cache

Method: Polynomial Pearson's Correlation **Model**: Lasso **Alpha**: 0.01 **Degree**: 3 **PCA components**: 5 R^2 **Score**: 74%

Formula: Too large

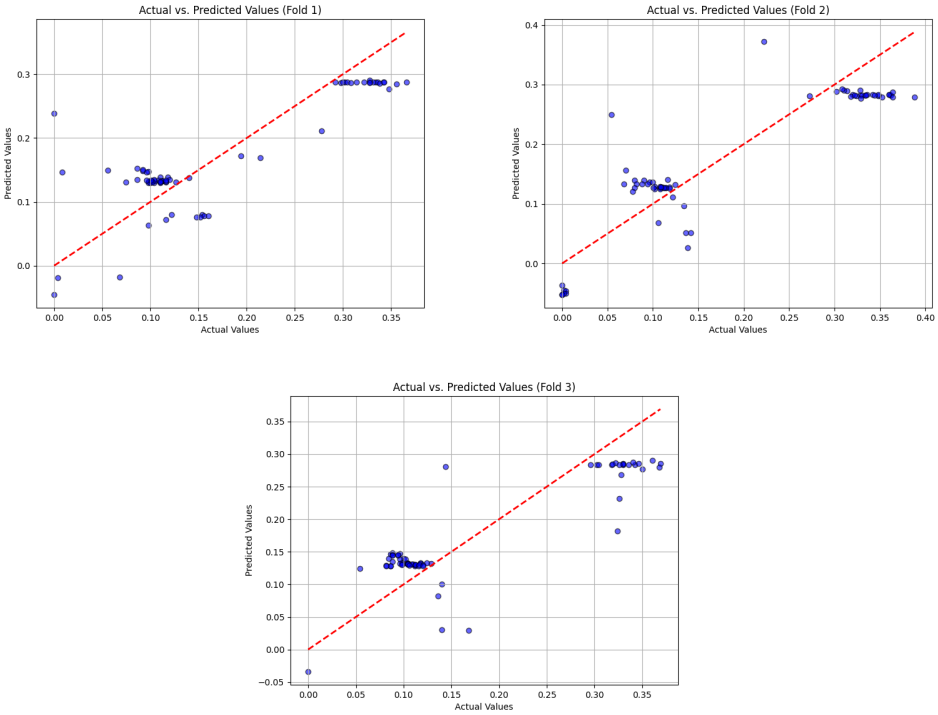

Figure 6.21: SDC-AVF, Data Cache, 5 maximum features, K-fold validation

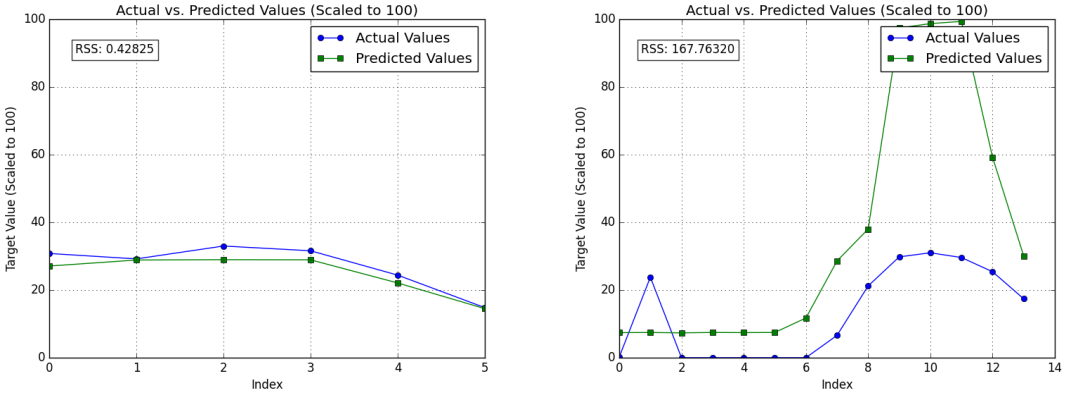

Figure 6.22: SDC-AVF, Data Cache, 5 maximum features, Test Programs

PRIM

Threshold: 30%

Table 6.21: SDC-AVF, Data Cache, 5 maximum features, PRIM statistics

Metric	Value
Coverage	0.959184
Density	0.912621
Mass	0.300292
Mean	0.912621

Table 6.22: SDC-AVF, Data Cache, 5 maximum features, PRIM box

Metric	Min	Max	QP Values
system.cpu.dcache.ReadReq.misses::cpu.data	946	34202	$4.15e-24$
system.cpu.ipc	2.1	4.3	$4.63e-03$
system.cpu.dcache.WriteReq.accesses::cpu.data	20480	3192842	$3.44e-01$
system.cpu.dcache.WriteReq.misses::cpu.data	362	33602	$4.62e-01$

6.2.2.3 L1 Instruction Cache

Method: Linear Procedure **Model**: Linear Regression **Degree**: 1 **PCA components**: 3 R^2 **Score**: 75%

Formula:

$$
\begin{aligned}
&- 0.01292 \cdot system.cpu.icache.demandMisses :: cpu.inst \\
&- 0.01866 \cdot system.cpu.icache.tags.totalRefs \\
&- 0.02394 \cdot system.cpu.ipc \\
&+ 0.03117
\end{aligned}
$$

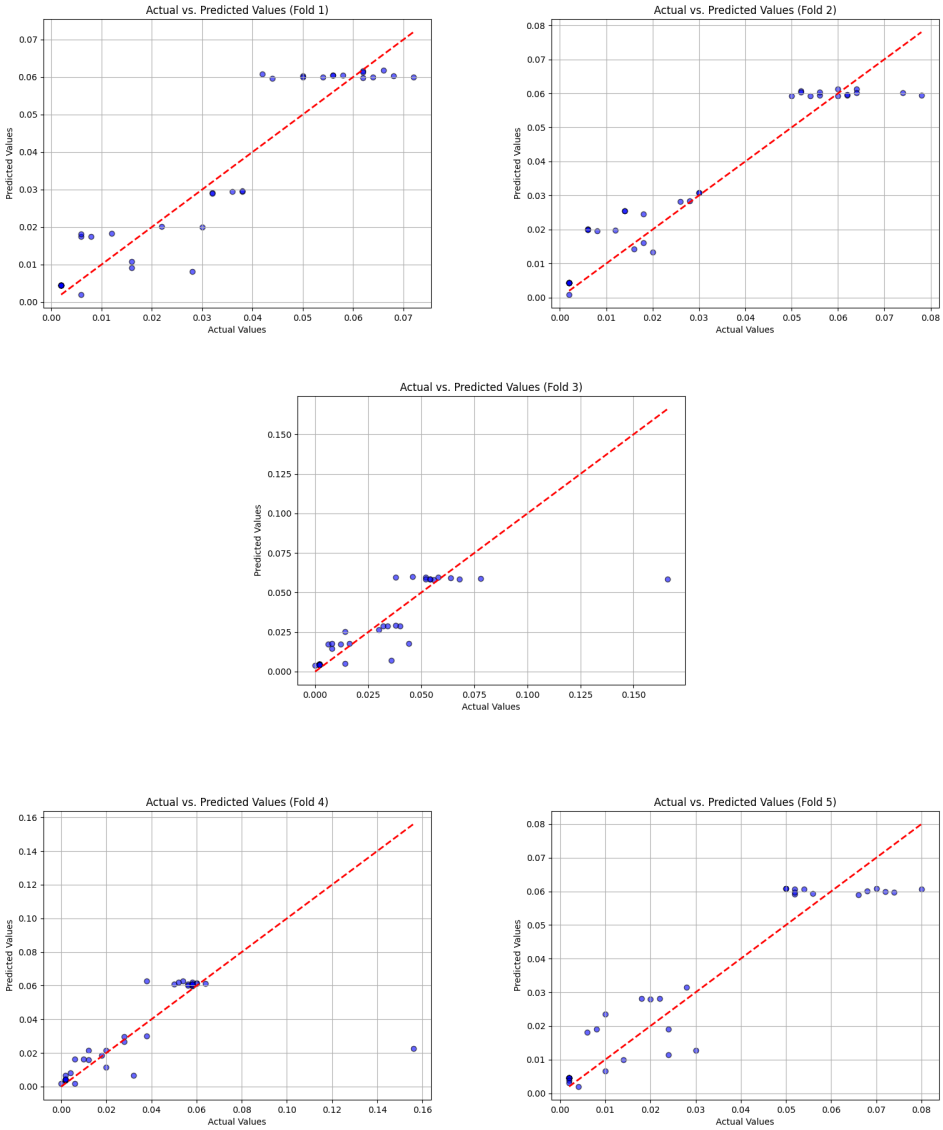

Figure 6.23: SDC-AVF, Instruction Cache, 5 maximum features, K-fold validation

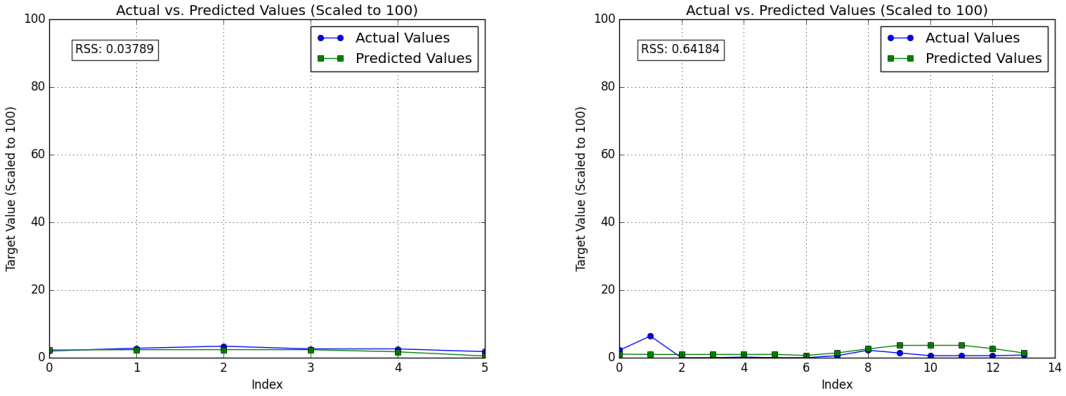

Figure 6.24: SDC-AVF, Instruction Cache, 5 maximum features, Test Programs

PRIM

Threshold: 10%

Table 6.23: SDC-AVF, Instruction Cache, 5 maximum features, PRIM statistics

Metric	Value
Coverage	1
Density	0.333333
Mass	0.032967
Mean	0.333333

Table 6.24: SDC-AVF, Instruction Cache, 5 maximum features, PRIM box

Metric	Min	Max	QP Values
system.cpu.icache.tags.totalRefs	2346980	2358469	$1.05e-01$
system.cpu.icache.demandMisses::cpu.inst	30979	163825	$3.00e-01$

6.3 Conclusion

The results undoubtedly indicate a conservative and moderate correlation between the features of each component and their corresponding target value (total-AVF or SDC-AVF).The R^2 scores are high enough to support this assumption, even when a maximum of five features are used in the regression procedure (they all exceed 61%). These maximum of five features regression models may be preferable, as they have reduced risk of overfitting and allow for a more straightforward hardware implementation. Additionally, the results from the PRIM align well with the performance of the corresponding regression models as expected.

However, while in many statistical analyses an R^2 of this magnitude would be considered more than enough, AVF prediction requires greater caution due to the critical nature of this metric. Despite the seemingly high R^2 values, the regression procedures often fail to capture the noise in the train and test execution intervals, leading to significant deviation in predictions which can lead to less informed runtime decisions for mitigation like the RMT enabling/disabling know mentioned at the beginning of the thesis. In many cases, the models fall extremely off, indicating difficulties in generalizing across different workloads and execution conditions. For example, these figures: 6.4 6.10 6.16 6.22 indicate poor generalization of the regression models, despite the fact that the same models may perform well on the training data. Also, it is important to keep in mind that the results shown in the test programs are scaled to 100, which may give the impression of greater variation than what actually exists. In reality, the raw values show much lower variation, meaning that even small differences in the scaled results may correspond to significant deviations. Therefore, considering the tested **methodologies**, **hardware components**

and configuration and **input data** (intervals of program execution), the results from the test programs, along with a close examination of the k-fold validation, suggest that the correlation is not strong enough to support any kind of implementation of reliability detection mechanism or AVF-aware execution strategies.

Compared to other studies, these findings highlight the difficulties and limitations of using regression for the prediction of the AVF and point to the need of further research to improve the prediction accuracy.

6.4 Future Work

As of now, the field of runtime AVF prediction still remains mostly unexplored. Finding the optimal prediction formula for AVF that balances performance and reliability is a crucial challenge in processor design that remains unanswered. Future research could explore deep learning techniques to improve AVF prediction accuracy, using neural networks to capture complex patterns in hardware behavior. Also, different workloads that specifically overload a component are yet to be studied. Such benchmarks could improve correlation analysis and add more variation to the input data.

To support real-time AVF estimation, the design and implementation of dedicated hardware counters that track error vulnerability at runtime is essential. These counters should act as hardware proxies and monitor the microarchitectural events relevant to AVF, allowing for more responsive and accurate AVF prediction.

With an unified SFI framework like gem5-MARVEL, there are many observational studies that can be done as well. For example, applying this AVF estimation framework to different architectures, such as ARM and x86, or even to different hardware configurations could provide broader insights into processor reliability across various computing environments.

Finally, an interesting future direction is to examine the interplay between AVF, power efficiency and security, ensuring that reliability improvements do not introduce new flaws in AVF-aware execution.

ABBREVIATIONS - ACRONYMS

MTTF	Mean Time To Failure
MTBF	Mean Time Between Failures
MTTR	Mean Time To Repair
AVF	Architectural Vulnerability Factor
SDC	Silent Data Corruption
RMT	Redundant Multi-Threading
RISC	Reduced Instruction Set Computing
CISC	Complex Instruction Set Computing
ACE	Architecturally Correct Execution
SFI	Statistical Fault Injection
ISA	Instruction Set Architecture
SE	Syscall Emulation
FS	Full System
PRIM	Patient Rule Induction Method
PCA	Principal Component Analysis

BIBLIOGRAPHY

[1] gem5. https://www.gem5.org/.

[2] Patient rule induction method for python. https://github.com/Project-Platypus/PRIM.

[3] Single instruction multiple data. https://www.sciencedirect.com/topics/computer-science/single-instruction-multiple-data.

[4] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[5] Jeremy Bennett. mibench. https://github.com/embecosm/mibench, 2012.

[6] IEEE Spectrum Blog. Impact of cosmic rays in hardware. https://spectrum.ieee.org/cosmic-ray-failures-of-power-semiconductor-devices, 2019.

[7] Benjamin P. Bryant and Robert J. Lempert. Thinking inside the box: A participatory, computer-assisted approach to scenario discovery. *Technological Forecasting and Social Change*, 77(1):34–49, 2010.

[8] Odysseas Chatzopoulos, George Papadimitriou, Vasileios Karakostas, and Dimitris Gizopoulos. Gem5-marvel: Microarchitecture-level resilience analysis of heterogeneous soc architectures. pages 543–559, 03 2024.

[9] Jerome Friedman and Nicholas Fisher. Bump hunting in high-dimensional data. *Statistics and Computing*, 9, 04 1999.

[10] Xin Fu and José A. B. Fortes. Sim-soda : A unified framework for architectural level software reliability analysis. 2006.

[11] A.D. George. An overview of risc vs. cisc. In *[1990] Proceedings. The Twenty-Second Southeastern Symposium on System Theory*, pages 436–438, 1990.

[12] Randy Fish Jyotika Athavale. Examining silent data corruption: A lurking, persistent problem in computing. https://www.synopsys.com/blogs/chip-design/what-is-silent-data-corruption-sdc.html, 2024.

[13] Manolis Kaliorakis, Sotiris Tselonis, Athanasios Chatzidimitriou, Nikos Foutris, and Dimitris Gizopoulos. Differential fault injection on microarchitectural simulators. In *2015 IEEE International Symposium on Workload Characterization*, pages 172–182, 2015.

[14] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *2009 Design, Automation Test in Europe Conference Exhibition*, pages 502–506, 2009.

[15] Bin Li, Lide Duan, and Lu Peng. Efficient microarchitectural vulnerabilities prediction using boosted regression trees and patient rule inductions. *IEEE Transactions on Computers*, 59(5):593–607, 2010.

[16] Daniel McFarlin, Charles Tucker, and Craig Zilles. Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism? volume 48, pages 241–252, 03 2013.

[17] S.S. Mukherjee, M. Kontz, and S.K. Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 99–110, 2002.

[18] George Papadimitriou and Dimitris Gizopoulos. Demystifying the system vulnerability stack: transient fault effects across the layers. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 902–915. IEEE Press, 2021.

[19] George Papadimitriou and Dimitris Gizopoulos. Avgi: Microarchitecture-driven, fast and accurate vulnerability assessment. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 935–948, 2023.

[20] Joel Emer Steven K. Reinhardt Shubhendu S. Mukherjee, Christopher Weaver and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36*, pages 29–40. IEEE, 2003.

[21] Kristen R. Walcott, Greg Humphreys, and Sudhanva Gurumurthi. Dynamic prediction of architectural vulnerability from microarchitectural state. *SIGARCH Comput. Archit. News*, 35(2):516–527, June 2007.

[22] Andrew Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, University of California, Berkeley, 2016.