3-21-2014

A Regression Approach to Execution Time Estimation for Programs Running on Multicore Systems

Mohammad Alshamlan
malsh002@fiu.edu

DOI: 10.25148/etd.FI14040809

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

A REGRESSION APPROACH TO EXECUTION TIME ESTIMATION FOR PROGRAMS RUNNING ON MULTICORE SYSTEMS

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Mohammad Alshamlan

2014

To: Dean Amir Mirmiran
College of Engineering and Computing

This thesis, written by Mohammad Alshamlan, and entitled A Regression Approach to Execution Time Estimation for Programs Running on Multicore Systems, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this thesis and recommend that it be approved.

Hai Deng

Sakhrat Khizroev

Gang Quan, Major Professor

Date of Defense: March 21, 2014

The thesis of Mohammad Alshamlan is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2014

DEDICATION

I would like to dedicate the thesis to my dearest mother Ms. Suhailah Alrukayess and loving father Mr.Ahmad Alshamlan. Without their love, understanding, support, and encouragement, the completion of this endeavor would never have been possible.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor Dr. Gang Quan. I feel fortunate and blessed to have an advisor like Dr. Quan, who gave me constant guidance, personal attention, suggestions and endless encouragement during the my master study. I truly admire his perseverance, depth of knowledge and strong dedication to students and quality research.

I am very grateful to my Ph.D. committee members: Dr. Hai Deng and Dr. Sakhrat Khizroev for their thoughtful insights and suggestions in improving my research. I am extremely proud to have such a wonderful and knowledgeable people serving on my thesis committee.

I am thankful to the staff of ECE department at FIU, specially to Mrs. Maria Benincasa, Mrs. Pat Brammer and Mrs. Ana Saenz for their great commitment to student services.

Next, I would like to thank my lab mates at ARCS lab for creating a wonderfully collaborative and enriching work environment filled with fun and laughter. I am confident that our friendship and cooperation will go a long way.

Finally, and above all, I want to thank my family for their unlimited love, faith, encouragement, blessings and prayers. My life-long gratitude go to my dearest mother Mrs. Suhailah Alrukayess and my loving father Mr.Ahmad Alshamlan for all the love. My parents are my true inspiration.

ABSTRACT OF THE THESIS

A REGRESSION APPROACH TO EXECUTION TIME ESTIMATION FOR PROGRAMS RUNNING ON MULTICORE SYSTEMS

by

Mohammad Alshamlan

Florida International University, 2014

Miami, Florida

Professor Gang Quan, Major Professor

Execution time estimation plays an important role in computer system design. It is particularly critical in real-time system design, where to meet a deadline can be as important as to ensure the logical correctness of a program. To accurately estimate the execution time of a program can be extremely challenging, since the execution time of a program varies with inputs, the underlying computer architectures, and run-time dynamics, among other factors. The problem becomes even more challenging as computing systems moving from single core to multi-core platforms, with more hardware resources shared by multiple processing cores.

The goal of this research is to investigate the relationship between the execution time of a program and the underlying architecture features (e.g. cache size, associativity, memory latency), as well as its run-time characteristics (e.g. cache miss ratios), and based on which, to estimate its execution time on a multi-core platform based on a regression approach. We developed our test platform based on GEM5, an open-source multi-core cycle-accurate simulation tool set. Our experimental results show clearly the strong relationship of the program execution time to architecture features and run-time characteristics. Moreover, we developed different execution time estimation algorithms using the regression approach for different programs with different software characteristics to improve the estimation accuracy.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Execution time estimation is important in design of micro-processor based computing systems. It provides the basic guidelines for selections of processors and other hardware components for the systems. It also critical for resource management unit when scheduling program execution to meet the design constraints and optimize the system performance and other optimization goal. Estimating a program execution time is particularly critical in design of real-time systems [8, 30, 22]. Real-time systems require more than delivering accurately produced computational results. They also require tasks to meet their deadlines because, for applications such as medical and avionic applications, missing a deadline can have catastrophic consequences, such as loss of life or plane crash [21, 19].

In this chapter, we first introduce the challenges of the execution time estimation problem. We then discuss the motivation and hypothesis of our research. Finally, we summarize our contributions.

1.1 Execution Time Estimation

Execution time estimation is a hard problem [40]. First, the program execution times vary not only with different architectures and specific hardware configurations [25]. The variations become larger and larger as processor architectures become more and more complicated today [7, 41]. Also, the program execution times vary with different software characteristics of the programs. Different inputs, loop counts, recursive functions, and execution paths all contribute to significant execution time variations.

From the architecture perspective, the challenges for estimating the execution time for a program on a single processor platform are mainly related to non-deterministic natures in cache, pipeline, and out-of-order execution [34]. While there have been substantial amount of work for execution time estimation on single processor platform, these approaches can be largely categorized into two groups: the static (or analytical) approach and the measurement based approach [40].

The static approaches bound the execution time of a program by analyzing the possible control flow of a program and then combining it with a hardware architecture model, such as value analysis which assigns the code addresses statically to combine registers and local variables [38]. In a static approach, to explore all the possible path can be challenging and time consuming. For example, Li et al. proposed to bound the worst case execution time by formulating the problem as an integer linear program, which can only be applicable for small size of problem due to the complexity when solving the problem [28]. In addition, to construct an appropriate abstract model for processor behavior including memory hierarchy, data buses, and I/O devices can also be challenging and time consuming [40].

The measurement based approaches (e.g. hardware tracing mechanisms), on the other hand, estimate program execution times by running programs on particular hardware or simulation platforms [44, 40]. The measurement based approach helps to identify the architecture impacts in an intuitive and straightforward manner, without going through the analytical analysis based on abstract architecture models. However, for measurement based approaches, to exhaustively explore all execution paths is usually impossible. To determine the worst-case path or states of architecture components is hard and normally impossible as well.

This problem becomes even more challenging for multi-core systems because hardware resources such as cache and buses are shared by multi cores, which substantially increases the execution time variances for programs running on multiple cores [36]. For example, Yan and Zhang proposed an analytical approach for multi-core systems that incorporates shared caches for the execution time analysis [41]. To ease the problem in estimating the execution time of each real-time task in an architecture with shared L2, they had to simplify their model by assuming that all cache access to L1 data caches are cache misses, and all accesses to L1 instruction caches are cache hits [25]. In addition, the execution synchronization needs among different parallel programs also exacerbate this problem [18, 16, 26]. As a result, while multi-core processors have become ubiquitous, the challenge of estimating the execution time of a program has increased significantly.

In this research, we develop a new coarse-grain approach for execution time estimation for programs running on multi-core platforms. The rationale is that the traditional execution time estimation approaches, i.e. the static or measurement-based approaches, are targeted at on one specific architecture type and configuration usually are very timing consuming in design space exploration for computing system design. Instead, we intend to identify the relationship between the execution time and hardware architecture configurations to facilitate the fast execution time estimation for programs running on multi-core platforms.

1.2 The Research Hypothesis and Our Research

In this research, we intend to develop computational efficient techniques to estimate the program execution time quickly to facilitate the design space exploration in system design. This research hypothesizes that there is a close relationship between the execution time and hardware architecture configurations, and regression algorithm can effectively capture this relationship [15, 17, 4]. In addition, we believe that the execution time estimation techniques should take into considerations not only the architecture features, but also software characteristics as well. As a result, in our research, we classify programs into different categories based on their software characteristics, and develop estimation models for each category separately in order to improve the estimation accuracy.

To capture the relationship between program execution times with different architecture features, we need to simulate program execution times under a larger variety of different hardware architecture configurations. To facilitate our research, we employed a cycle accurate architecture simulator called GEM5 [9, 20, 42].

The cycle accurate architecture simulator GEM5 is a platform for research in the areas of operating systems, compilers, and computer architecture [20, 10, 22, 46]. Simply put, GEM5 encompasses system-level architecture as well as processor micro-architecture.

We also chose the popular benchmark in our research. The benchmarks adopted in our research come from Malardalen's benchmark project [2, 32, 21]. Malardalen benchmarks are collaborative programs developed and maintained by multidisciplinary researchers commonly used for Worst Case Execution Time (WCET) estimation when designing real-time devices [8, 45, 31].

A program, for real-time systems, has unique software characteristics, such as: always single path program, program contains loop, program contains nested loops, program uses arrays or matrices, program contains recursion, or a program uses floating point calculation. The Malardlen's benchmark takes these software characteristics into consideration, developed with the purpose to assist system designer to analyze and examine the worst case execution times for programs in different types of applications, which fits our research needs very well.

1.3 Significance of This Research and Our Contributions

Execution time estimation is critical in design of computing systems, especially real-time systems when timing is as important as logical correctness [13, 29, 43, 27]. To explore a large variety of design alternatives, the system level design of real time computing systems demands effective and efficient computational methods for estimating execution time [39, 19, 46, 11, 35, 5].

The research contributes by verifying and disclosing the close relationship between program execution time and multi-core hardware architecture configurations and, based on which, to develop an effective and efficient way to estimate program execution under different multi-core architectures. The approach classifies commonly used software characteristics in categories that allow us to compare the hardware architecture efficiently.

By pinpointing each set of these Hardware/Software (HW/SW) groups in the system, we can analyze the cause and effects in the system performance and the execution time for different hardware architectures [39, 37, 11].

The contributions of this research are the following:

1. Verifying and disclosing the close relationship between program execution times and architectural settings;

2. Using a regression approach to develop a computation efficient method to estimate execution times for programs with different software characteristics;

3. Investigating the effectiveness of the proposed approach using architecture level cycle accurate simulators and well-known benchmark.

1.4 Thesis Organization

Chapter 2 provides background information about the simulation platform and the Malardlen benchmark. In chapter 3, we discuss our efforts in the development of the integrated simulation platform and also the software characteristics based on which we categorize the benchmark programs. Chapter 4 presents our simulation results and validation results. Specifically, we present the analytical formula obtained through the regression method for program in each category first. We then present our validation results. Chapter 5 concludes the thesis and summarizes the thesis contributions.

CHAPTER 2
Background

In this chapter, we discuss our efforts in the development of our simulation environment that is flexible and customizable, which can be used to effectively conduct a large number of simulations. We also discuss the benchmarks, i.e. the Malardlen, that are used for deriving the execution time estimation.

2.1 The Cycle Accurate Architectural Simulator

The cycle accurate architectural simulator is a platform for research in the areas of operating systems, compilers, and computer architecture [20]. Simply put, the cycle accurate architectural simulator encompasses system-level architecture as well as processor microarchitecture. The goals are to have an intuitive approach, and they can be used to validate the developed system result for an actual system. The desirable features of the cycle accurate architectural simulator usually are the following:

1. Useful timing measurements (cycle-accurate)
2. Support for fully functional OSes (full-system)
3. Auxiliary features for specific experiments
4. Support for useful hardware platforms and instruction sets
5. Openness to modifying microarchitectural features
6. Availability of technical support

For the purpose of this research, the cycle accurate architectural simulator must be able to run hundreds of thousands of experiments back-to-back. Each experiment must be unique to all the other executed experiments, and that experiment output must be stored in a unique file naming convention because the file name is used as an ID.

There are other features that are not required, but they are preferable to have. Community collaboration is important, and there are aspects that lead to an excellent community collaboration. Usually, the cycle accurate architectural simulators licensing terms and code quality can affect the community collaboration [24].

There are some open-source licenses that can be too restrictive for industry collaborations in the developer community. The licenses that were used for the simulator are not that important for this thesis. However, their effects can limit some contributions from a developer, especially from an industrial setting [45]. Therefore, looking at the cycle accurate architectural simulator licenses is not the focus of this research. Nonetheless, they have been looked into to make sure that the used cycle accurate architectural simulator has been developed from both industry and academia.

Also, the cycle accurate architectural simulator's internal code can limit community collaborations. A poor code quality and lack of modularity can be intimidating to developers [14, 29, 43]. If a simulator lacks some features, adding this feature can be challenging, and can even consume precious time that a developer does not have.

Therefore, these two features are not important, but they are taken into consideration because the code quality and the license of a cycle accurate architectural simulator affect community collaborations. There is a wide variety of cycle accurate architectural simulators that meet these requirements. Due to the overwhelming choices and flavors of these simulators, we evaluated three cycle accurate architectural simulators as shown in this section.

2.1.1 SimpleScalar

At first, SimpleScalar was chosen for the research because it is an open-source architectural simulator. These are some of SimpleScalar capabilities:

First, because SimpleScalar is an open-source, it means SimpleScalar can be modified. In fact, there are several successful stories that show a sophisticated architectural simulator is built from SimpleScalar [3]. For example, MASE and Wattch use SimpleScalar as their base for distinctive and much more concentrated functions and features. In the case of MASE, it gets a ground-up enhancement from SimpleScalar to deliver a powerful micro-architectural modeling that is even more advanced than SimpleScalar can offer. Wattch focuses on a different area, which is simulating the power consumption for a given microprocessor. Wattch developers did not see a need to reinvent the wheel, so they have adapted an open-source tool, which was SimpleScalar, and started adding the missing functions [12].

The second desirable feature from SimpleScalar is a wide variety of Instruction Set Architecture (ISA) [6]. SimpleScalar supports common commercial ISA, such as: Alpha, ARM, and x86. In addition, SimpleScalar shines for an academic ISA that can be used as an academic research-oriented tool. This academic ISA, which SimpleScalar supports, is not the common MIPS ISA, but SimpleScalar uses Portable Instruction Set Architecture (PISA) [3]. PISA has a its own compiler, which supports a common compiler such as GNU GCC. Unfortunately, there is not a strong industry adaptation for PISA because PISA's objective is to be used for educational purposes.

With these key features, such as flexibility, openness, Multi-ISA support, and community collaborative work, SimpleScalar has been used as a preliminary prototype. SimpleScalar's internal architecture is shown in Figure 2.1.

The enhancement of SimpleScalar has begun, but shortly thereafter, there were several limitations that became unavoidable to achieve the research goal. One of the drawbacks was that SimpleScalar lacks arbitrary instruction restart. The only restart that SimpleScalar can perform is branchescan restart, which is not helpful for the research platform. SimpleScalar is not chosen because the platform that is being developed must have the ability to run contiguous experiments with no human intervention through the automated test engine as discussed in chapter 3.

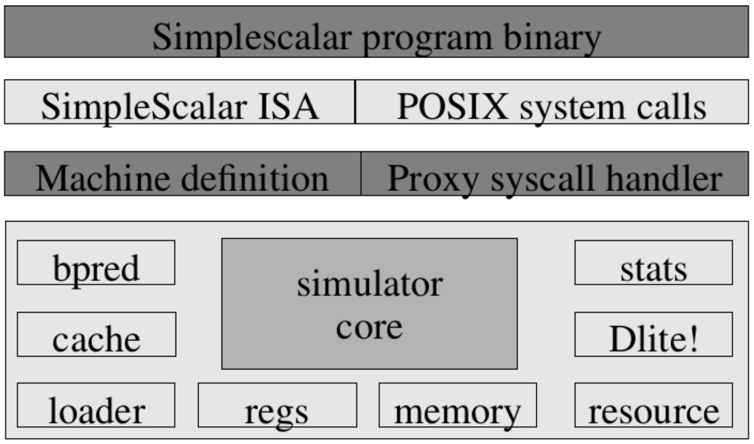

Figure 2.1: SimpleScalar's Internal Architecture [6]

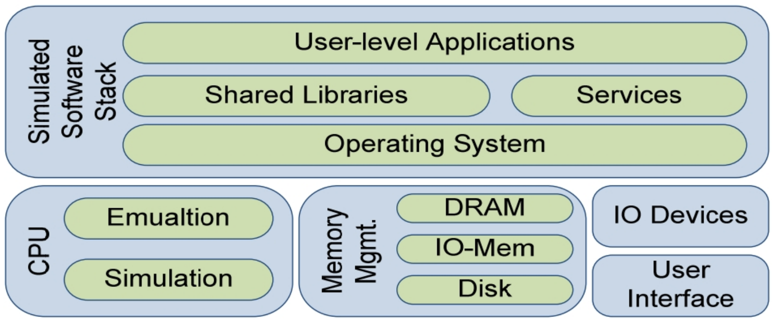

Figure 2.2: MARSS' Internal Architecture [33]

2.1.2 MARSS

MARSS is a simulator that has an arbitrary restartable mechanism that gives a sequential unique experiment without a human intervention. MARSS supports just x86 ISA, which limits the research scope [33]. Figure 2.2 illustrates MARSS' internals.

The research platform can use MARSS to do a sheer number of experiments. The combination between the research platform and MARSS gives the ability to manipulate a variety of hardware configurations, such as: cache associativity, TDMA bus width, rearranging memory hierarchy, device latencies, and system cycles.

However, MARSS has a drawback, even if MARSS is a good candidate as a cycle accurate architectural simulator for the research platform. The capabilities of MARSS are appealing, but its speed of executing an experiment is not. MARSS is slow to perform

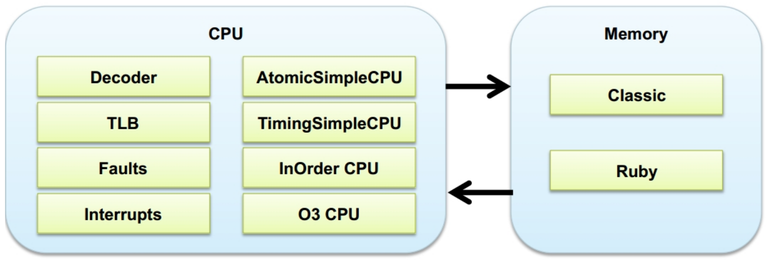

Figure 2.3: GEM5's Internal Architecture [9]

an experiment, so this drawback challenges the time constraints of the research.

This problem is hard to spot, but running $100,000$ samples of experiments will take MARSS more than a month just to do all the parameter combinations. This latency is not acceptable, so MARSS is a great architectural simulator, but its speed would leave the research in jeopardy.

2.1.3 GEM5

GEM5 is an emerging platform from M5 and GEMS, which both are a flexible simulation framework. The inheritance of this flexibility allows GEM5 to evaluate diverse architectural design ideas with a rich support for OS facilities, such as including IO and networking [1]. However, these features have been achieved with MARSS, but GEM5 is much faster than MARSS, and that is why the research is using GEM5 as a base simulator. Figure 2.3 illustrates GEM5's internals.

There are other functions and features where GEM5 surpasses MARSS capabilities. MARSS supports just one type of ISA, but GEM5 supports the following: Alpha, ARM, SPARC, MIPS, POWER and x86 ISAs. One of the aspects that have been discussed and looked into is the simulator license. GEM5 supports a BSD-based license which does not have awkward legal restrictions [9]. The source code is available to all researchers, regardless of their uses, that allows GEM5 to have a good mix between academic and industry collaborations [24]. In fact, these are some of the major GEM5 contributors: AMD, ARM, HP, MIPS,MIT, Texas, and Wisconsin [1, 43, 14, 24].

Because GEM5 is a full emergence of M5 and GEMS, the developer community of GEM5 has many researchers' attention. M5 and GEMS have been used in hundreds of publications and have been downloaded tens of thousands of times [1]. Many of these researchers are already familiarized with GEM5 because GEM5 supports both M5 and GEMS syntaxes [14].

2.2 About GEM5 Simulator

In this section, some basic fundamentals in GEM5 are tested. We have examined some potential functions and features in GEM5 for enhancement purposes before integrating them into the research platform. The GEM5 accuracy has been looked into, and the drawbacks with this approach are discussed.

Note, all the source codes of implementation are provided in the appendices, so the reader can replicate the research platform.

2.2.1 Coding Style of GEM5

GEM5 is a simulator that is built from two previous simulators which are M5 and GEMS. By writing a comprehensive hardware architecture design, GEM5 executes the hardware architecture deign with cycle accuracy.

Because M5 and GEMS have their own coding syntaxes, GEM5 accepts either coding syntax. The coding style that is used in this thesis is M5 and not GEMS. The reason for choosing M5 rather than choosing GEMS is simply due to the following aspects. M5's support does not always mean GEMS support [9]. The emergence of M5 and GEMS is not completed yet. There are areas in GEM5 that are not fully emerged, so the developer needs to choose a specific coding syntax. In other words, the developer has to choose either M5 or GEMS for the non-emerged areas. Unfortunately, that is what happened to this research.

The objective of this research is to allow the research platform to manipulate the hardware configurations of GEM5 through M5. In actuality, that means the research platform manipulates the M5's SimObjects. SimObjects are located in modules of concrete hardware components called SimObjects, and the SimObjects of M5 are the following: diverse CPU models, PCI, NICs, IDE controller, a frame buffer, DMA engines, UARTs, and interrupt controllers. These SimObjects provide a highly configurable simulation framework, multiple cache coherence, and multiple ISAs, such as: ARM, ALPHA, MIPS, Power, SPARC, and x86. SimObjects are discussed in detail in the next subsection.

2.2.2 GEM5 Internals

Knowing the internals of GEM5 helps developing a tight integration, so to accomplish this goal there are questions that should be answered in this subsection, which are the following. How is GEM5 implemented? Where is the main() function of GEM5? What is the type of object orientation that GEM5 uses? What are the programming languages that GEM5 supports? What are the domain-specific languages that GEM5 supports? What are the standard interfaces that GEM5 supports?

First, GEM5 adopts an object-oriented design methodology. One of the advantages is that the developer does not need to understand the entire source code for modifications or contributions. For this research, object-oriented design allows to enhance a particular section of a code without going through a tedious modification of the whole source code. The reason of this accomplishment is because object-oriented design favors modularity, so the developer just focuses on the module that needs an adjustment.

Naming a few composable objects, they are: cache hierarchy, bus configurations, multi-core, and system clock. The major simulation components of GEM5 are located in models of concrete hardware components called SimObjects. What is noticeable of SimObjects is that, they share the same common behaviors in regards of configuration, initialization, statistics, and serialization. That means the method of controlling one

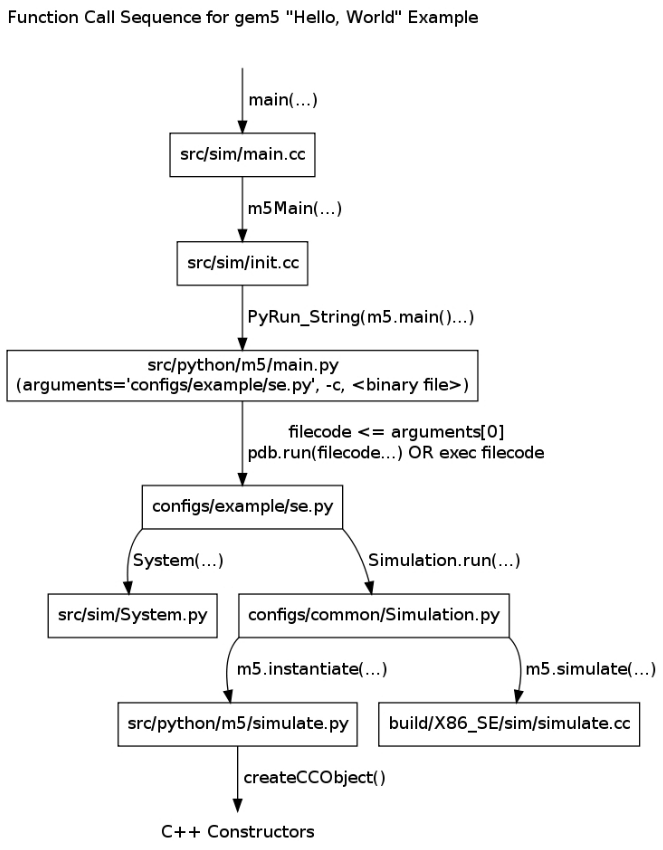
Function Call Sequence for gem5 "Hello, World" Example
main(...)
src/sim/main.cc
m5Main(...)
src/sim/init.cc
PyRun_String(m5.main()...)
src/python/m5/main.py
(arguments='configs/example/se.py', -c, <binary file>)
filecode <= arguments[0]
pdb.run(filecode...) OR exec filecode
configs/example/se.py
System(...)
Simulation.run(...)
src/sim/System.py
configs/common/Simulation.py
m5.instantiate(...)
m5.simulate(...)
src/python/m5/simulate.py
build/X86_SE/sim/simulate.cc
createCCObject()
C++ Constructors

Figure 2.4: A Startup Procedure In GEM5 [9]

SimObject can be applied to any other SimObject. These SimObjects can be system cores, caches, or system interconnection, but they also can be an abstract entities, such as a workload or a process context-switch for the design architecture.

Second, Python implementation in GEM5 is just 15% of all the source code, so there are limitations for using Python instead of C++. However, the SimObjects that are not implemented in Python are lower level designs, such as creating a new ISA. Simply put, most of C++ SimObjects that are not duplicated in Python SimObjects are not needed for this research.

Also, the main function of GEM5 is written in Python, and start-up code is built-in to the main function. The simulator begins executing Python code almost immediately on start-up. Figure 2.4 illustrates a start-up procedure.

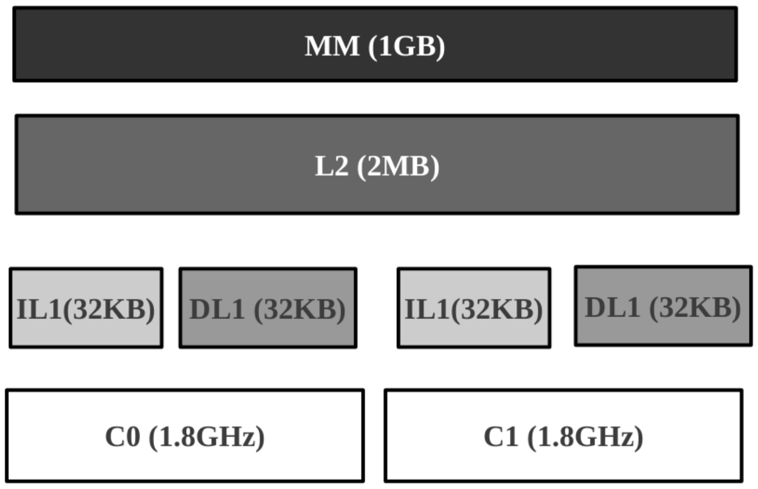

Figure 2.5: A Dual Core Architecture

Third, GEM5 supports Domain-Specific Languages (DSLs), which are instruction sets and cache coherence protocols. These two DSLs are important because they can provide a powerful and concise way to express a variety of solutions that can be from the logic design layer to the compilation (compiler) layer. In this research, we are not defining a new Instruction Set Architecture (ISA), we just need to be aware of the compiler optimization because the program-flow can change the system performance, which affects the execution time of a program.

2.2.3 Ilustating GEM5 Capabilities

To understand GEM5, we need to have an example. The hardware design that is used in this example is shown in Figure 2.5.

Hardware Design

To make GEM5 creates this design as shown in Figure 2.5, we need to write a comprehensive hardware architecture design code in Python. The source code is shown in appendix A. Appendix A also shows an explanation of the source code implementation. The purpose for this simple code is to show the reader how GEM5 works and to show that

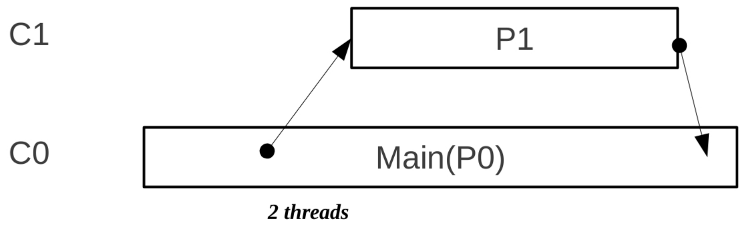

Figure 2.6: A Simple Two Threads Program for A Dual Core Architecture

GEM5 does not support Graphical User Interface (GUI). Every hardware architecture design must be written in programming language such as Python.

Software Design

Software instructions tell the hardware what to perform. These instructions are stored in the main memory by assumption. There are many methods to these instructions to be allocated in the main memory, but the common method is moving these instructions from a non-volatile memory to a volatile memory. For example, a hard disk, a non-volatile memory, has system boot instructions. The system wants to execute them to boot-up, so these instructions are moved to the main memory, volatile memory, for the processor to fetch them.

We would not use a kernel to utilize the hardware efficiently just for the sake of simplicity. The code that we are going to write has the necessary components to utilize the two cores of the hardware architecture design as shown in Figure 2.5. Although there are several methods to utilize the two cores, the used method is multithreading programming. The code flowchart is shown in Figure 2.6.

From Figure 2.6, we can see, there is a main thread that is running in core zero (C0). The main thread means that the main() function is located in that thread. Because the main() function makes the code executable, C0 can run the code without problems. However, core one (C1) would not be utilized at all unless the main() function designates a task to C1. In the Figure 2.6, we can see that we want to use C1, so we need to make the main function to mange and control C1. The used approach is through creating a new thread, and then mapping the new thread to C1. The source code of this program is shown in appendix B.

The reason for using C as the programming language in appendix B is because GEM5 executes a machine code. Therefore, the used programming language is irrelevant to GEM5. The coding language is considered to be a programmer personal preference. In our case, we chose C because C is effective and efficient in a low-level system programming. The compiler output is the machine-language that GEM5 understands and not the source code that is shown in appendix B. The code compilation is not discussed here, but the reader should keeps in mind the code that is shown in appendix B cannot run in GEM5 as is. As a result, using a complier is a must.

Experiment Setup

So far, we have developed the software in appendix B and the hardware in appendix A, but we have not integrated them together. The hardware that we have developed is Intel 64-bit machine, so we need to compile this code to be an Intel 64-bit machine code, this step is done as follows.

```
gcc -static 1_hello_c_2threads.c -o 1-hello-c-2threads.c
```

Listing 2.1: Compiling The Developed Software

A Linux environment would do it successfully on the terminal, but for this example the used command-line terminal is bash. After the compilation is successful, the hardware code is hard-coded for looking for a file named as same as the compiler output. The only step that is missing is to initialize the experiment as the following command:

```
build/X86/gem5.opt configs/example/Dual_core_Arch.py
```

Listing 2.2: Running GEM5

After GEM5 finishes, GEM5 would output a detailed cycle-accuracy report about the experiment. The output report is not shown because the output result needs more than 200 pages to be printed out, but the implementation steps are explicitly shown in the appendices. The reader can implement the steps in the appendices to retrieve the output result.

2.3 The Benchmarks

GEM5 runs any software without problems if the program has been compiled properly [1]. GEM5 executes the machine code, so the type of the program is irrelevant to GEM5, but the flow program matters. The program flow can be the same for different types of programs such as: system drivers, user applications, or benchmarks. Therefore, we do not need to focus on the type of the program, but we need to consider the program flow of each runnable program in GEM5.

If the type of a program is irrelevant to GEM5's outcome because the GEM5's outcome depends on the program flow, we can choose benchmarks instead of the others because their purpose is to measure the system performance. We can use benchmarks that have different program flows and then categorize them in different software characteristic sets. Different software characteristic sets are thoroughly discussed when deriving the execution time estimation, but before we start deriving these algorithms, we really need to understand the nature of benchmarks.

Benchmarks provide a metric for comparison among different hardware architectures [2]. Using a well-known benchmark can allow researchers to compare their results. Comparison can be broader and very useful when industry sector collaborates. Therefore, these popular benchmarks need to be examined and evaluated in the basis of which one is compatible with the research platform.

2.3.1 The Benchmark's Internals

In this subsection, we are addressing major benchmark aspects, which are: the program flow, loops, and structure of the program.

The Effect of A Program Flow

There are distinctive features that a benchmark may have. A program can have a single-path, or that program can have a multi-path program flow. In common cases, if a program does not support command-line arguments, the program most likely would not have a multi-path program flow. Because command-line arguments are commonly used as inputs, these inputs can change the program behavior.

For example, suppose a program asks to import data for analysis purposes. If a user inputs a file name as a data file in the command-line arguments, the program shows the analysis. On the other hand, when the user runs the program without importing the data file, the program asks the user where the data are stored, then the program shows the analysis. As we can see, there is a change in behavior. This change of program behavior is significant to the program flow.

A single-path program flow has its uses in real life application, but they are rare cases. Proportionally, multi-path programs dominate the released programs in the market because these programs can do more than a task. End-users and third-party developers favor multi-path programs because to them these multi-path programs are the same as a swiss-knife. Because the market-share of multi-path programs is much more than single-path programs, the research focuses more into a multi-path program flow than a single-path program flow.

The Effect of Loop

Loops have a strong role in the hardware performance because they can make the processor execute the same instructions multiple times. A program that contains loops should be considered different than a program that contains nested loops. Loops can be nested, which increases the system overhead.

Call Graph and Scope Hierarchy Graph

There are many methods to analyze the program flow, but we are going to use a Semantic Web sErvice Editing Tool (SWEET) to generate the program flow graph.

The SWEET tool can provide a call graph and a scope hierarchy graph. The call graph shows the building-blocks of program flow. The SWEET tool does not show every detail such as calls to functions and entries to loops. A scope hierarchy graph is a context sensitive graph showing calls to functions and entries to loops.

Call graph and scope hierarchy graph have common aspects which are the following. The root of the tree is always the main() function. The scope can be initiated from a function or a loop. An arrow from higher scope to lower scope represents a function scope or a loop scope.

2.3.2 Evaluating Existing Benchmarks

Benchmarks are useful in many areas in computer science and engineering, but there is not a well-known comprehensive benchmark [2]. That is why there is a sheer number of available benchmark suites.

Due to the lack of a popular comprehensive benchmark, the evaluation has to focus on a specific area, which is a low system benchmark. In fact, there are many benchmarks that we do not need, such as measuring the performance of hardware components other than the processor and the main memory. For example, measuring the performance of a hard disk, GPU, Ethernet, or other user IO devices would not be useful to this research. Useful information to this research are: the speed of the processor, throughput of the processor, and memory usage.

One of low-level benchmarks is Drystone [2]. Drystone benchmark evaluates the performance of various computing areas in the processor. For example, the ALU has a different data-path for integer calculations and for floating-point calculations, so Drystone has the ability to measure these differences in the ALU data-paths. Unfortunately, the evaluation of Drystone stops because Drystone is not compatible with GEM5 simulator.

SPEC CPU2000 is also a low level benchmark that is compatible with GEM5, and SPEC CPU2000 is generally used for measuring a system processor, memory subsystems, and methods of compilation. There are publications that used SPEC CPU2000 for embedded devices such as: automotive, digital imaging, digital entertainment, energy consumption, networking, office automation, and telecommunications. Hence, office automation can be a printer, so some researchers used SPEC CPU2000 to measure the performance of an actual printer.

Given their strong correlation to the embedded domain, SPEC CPU2000 has problems with driving the estimation algorithms. SPEC CPU2000 does not categorize specific software characteristics of each benchmark.

Malardalen benchmarks are designed in mind for execution time estimation analysis which means every small detail of benchmark software characteristics are given. Execution time estimation analysis looks into the program flow and the datapath extensively. For example, the analysis can measure how many cycles are needed for a specific datapath in the ALU, or it can trace the datapath for specific core in the processor. Malardalen benchmarks are compatible with GEM5, and because their software characteristics are well documented, they can be used for the research platform.

2.3.3 Software Characteristics of Malardalen Benchmarks

Malardlen benchmarks are written in C, which is the most common language for embedded devices and real-time systems. Malardlen benchmarks are shown in these Tables 2.1, 2.2, and 2.3 with their software characteristics.

Table 2.1: List A of Malardalen Benchmarks [2]

No.	Benchmark	Description	S	L	N	A	B	R	U	F
1	adpcm	Adaptive pulse code modulation algorithm		Yes						
2	bs	Binary search for the array of 15 integer elements		Yes		Yes				
3	cnt	Counts non-negative numbers in a matrix		Yes	Yes	Yes				
4	compress	Data compression program		Yes	Yes	Yes				
5	cover	Program for testing many paths	Yes	Yes						
6	crc	Cyclic redundancy check computation on 40 bytes of data	Yes	Yes		Yes	Yes			
7	duff	Using Duff's device from theJargon file to copy 43 byte array	Yes	Yes					Yes	
8	edn	Finite Impulse Response (FIR) filter calculations	Yes	Yes	Yes	Yes	Yes			
9	expint	Series expansion for computing an exponential integral function	Yes	Yes	Yes					
10	fac	Calculates the faculty function	Yes	Yes				Yes		
11	fdct	Fast Discrete Cosine Transform	Yes	Yes		Yes	Yes			
12	fft1	1024-point Fast Fourier Transform using the Cooly-Turkey algorithm	Yes	Yes	Yes	Yes				Yes

Table 2.2: List B of Malardalen Benchmarks [2]

No.	Benchmark	Description	S	L	N	A	B	R	U	F
13	fibcall	Simple iterative Fibonacci calculation, used to calculate fib(30)	Yes	Yes						
14	fir	Finite impulse response filter (signal processing algorithms) over a 700 items long sample		Yes	Yes	Yes				
15	insertsort	Insertion sort on a reversed array of size 10		Yes	Yes	Yes				
16	janne_complex	Nested loop program	Yes	Yes	Yes					
17	jfdctint	Discrete-cosine transformation on a 8x8 pixel block	Yes	Yes		Yes				
18	lcdnum	Read ten values, output half to LCD		Yes			Yes			
19	lms	LMS adaptive signal enhancement The input signal is a sine wave with added white noise	Yes	Yes		Yes				Yes
20	ludcmp	LU decomposition algorithm		Yes	Yes	Yes				Yes
21	matmult	Matrix multiplication of two 20x20 matrices	Yes	Yes	Yes	Yes				
22	minver	Inversion of floating point matrix	Yes	Yes	Yes	Yes				Yes

Table 2.3: List C of Malardalen Benchmarks [2]

No.	Benchmark	Description	S	L	N	A	B	R	U	F
23	ndes	Complex embedded code		Yes		Yes	Yes			
24	ns	Search in a multi-dimensional array		Yes	Yes	Yes				
25	nsichneu	Simulate an extended Petri Net		Yes						
26	prime	Calculates whether numbers are prime	Yes	Yes						
27	qsort-exam	Non-recursive version of quick sort algorithm		Yes	Yes	Yes				Yes
28	qurt	Root computation of quadratic equations	Yes	Yes		Yes				Yes
29	select	A function to select the Nth largest number in a floating point array		Yes	Yes	Yes				Yes
30	st	Statistics program		Yes		Yes				Yes
31	statemate	Automatically generated code		Yes						
32	ud	Calculation of matrixes	Yes	Yes	Yes					

As we can see from these Tables 2.1, 2.2, and 2.3, there are 31 benchmarks. Also, these Tables 2.1, 2.2, and 2.3 have columns are denoted as: **S**, **L**, **N**, **A**, **B**, **R**, **U**, and **F**. Each letter of these columns means a specific software characteristic as follows:

S: always a single path program.

L: Contains loops.

N: Contains nested loops.

A: uses arrays and/or matrices.

B: uses bit operation.

R: contains recursion.

U: contains unstructured code.

F: uses floating-point calculation.

Usually, a program for a real-time system would have more than one software characteristic. However, there are programs that just require a loop, which is just one software characteristic. These programs that just require a loop, for example, an adaptive pulse code modulation algorithm, simulate an extended Petri Net, and automatically generated code.

2.4 Summary

In this chapter, we have evaluated these cycle accurate architectural simulators: SimpleScalar, MARSS, and GEM5, and we have evaluated these benchmarks: Drystone, SPEC CPU2000, and Malardalen. After we decided on GEM5 and Malardlen benchmarks to cover the scope of the research, we have covered the simulator configurations and evaluated the software characteristic sets.

CHAPTER 3
Our Approach

In the previous chapters, we covered the hardware configurations and the common software characteristics, so in this chapter the implementation of research platform is discussed. The research platform is an automated test engine that is able to run and manage the needed experiments for supporting the research hypothesis.

3.1 The Automated Test Engine

There are three main elements in the automated test engine, which are:

1. Utilizing cycle accurate architecture simulator (GEM5)
2. Using benchmarks with different software characteristics (Malardalen)
3. Using robust regression (execution time estimation)

In this section, we are integrating each elements to the automated test engine, and we are enhancing the needed parts for having a tight integration.

3.1.1 Integrating GEM5

To integrate GEM5 to the automated test engine, we need to enhance the gray boxes that are shown in Figure 3.1, which are the topic of this subsection.

Making GEM5 Integrable

We already have developed in appendix A, a python script to control GEM5 hardware parameters and run a specific program on the hardware, but it lacks a command-line interface. Each configurable hardware parameter for the selected design architecture must be configured manually in the source code. For example, if a developer wants to

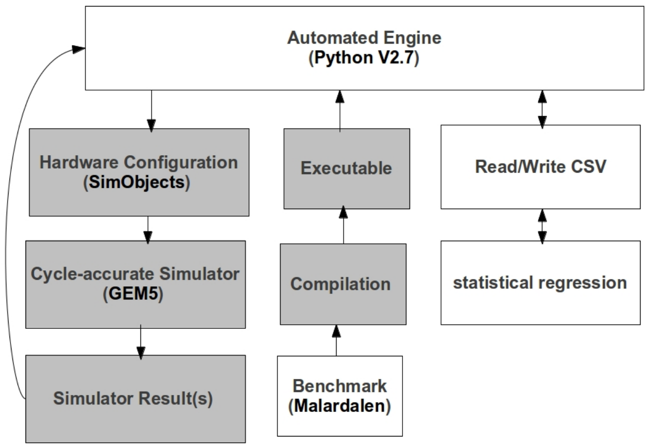

Figure 3.1: The Building-blocks of Enhanced Simulator

have 10 different hardware designs to compare, the developer needs to configure GEM5 code in appendix A by hands 10 times.

Human error plays a big role in this approach due to the fact that each experiments are run in different times, so it is easy for the researcher to get out of track from the current conducted experiment and the next potential ones.

GEM5 needs to be configured dynamically which means through the command-line interface. Human errors can make the results of regression algorithms to be inaccurate, so the automated test engine must manage the needed functionalities without human interventions. These enhancements are coded in the source code in appendix C, and the explanations of the code are also provided in appendix C.

The developer does not need to modify this source code for different hardware architectures or running different software characteristics. The command-line arguments are able to create different hardware architectures or running different software characteristics through the command-line and not through the source code. Therefore, GEM5 source code in appendix C can be automated.

To clarify the dynamic command-line configuration method, suppose we want to run the multithreaded hello world that we have developed in appendix B. The used machine needs to have 4 cores with 3GHz each. Its memory hierarchy as follows: cache block size is 256MB, stack is 128MB, main memory size is 0.5GB, TDMA bus slot is 4, L2 size is 8MB, L2 associativity is 8, DL1 size is 8KB, IL1 size is 8KB, and L1 latency is 9 cycle. The command to create this system is the following.

```
gem5.opt --outdir=/home/alshamlan/Desktop/WCET_Regression/raw_gem5_data/ --
    stats-file=1_experment  /home/alshamlan/Desktop/WCET_Regression/
    gem5_experimental_engine.py --num_cpus 4 --clock 3GHz --cmd /home/
    alshamlan/gem5/tests/test-progs/hello/bin/x86/linux/hello --block_size
    256 --Process_StackSize 128MB --phy_latency 40ns --Addr_Range 512MB --
    bus_slot 4 --l2_size=8MB --l2_assoc 8 --l2_hit_latency 90 --l2_mshrs
    110 --l2_tgts_per_mshr 14 --l2_write_buffers 10 --l1d_size 8kB --
    l1i_size 8kB --il1_assoc 1 --il1_hit_latency 9 --il1_mshrs 2 --
    il1_tgts_per_mshr 11 --dl1_assoc 4
```

Listing 3.1: The Enhanced Simulator

Further explanations of this experiment configurations are the following. First, we have a program is a multithreading program which would utilize more than one core. Second, the hardware configurations have been set up as the example states. Third, the output result would output to specific name which we define. If each experiment does not have a unique name, the simulator would overwrite the output result every time. Therefore, the automated test engine must give every runnable potential experiment a unique name to prevent result overwriting.

Summarizing the essential elements to make GEM5 integrable to the automated test engine, which are:

1. A program utilizes the hardware

2. The hardware can be configured dynamically without adjusting the source code

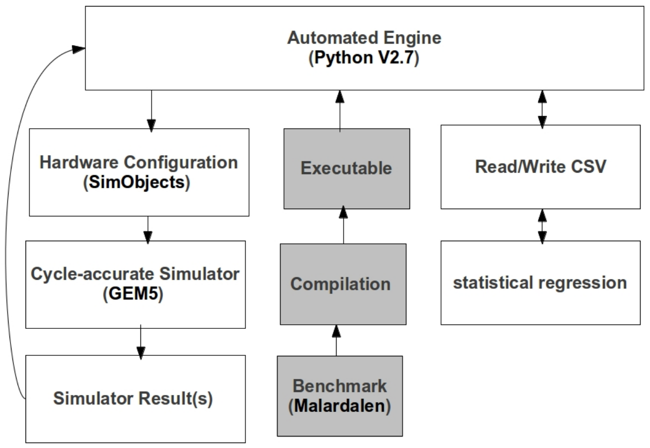

Figure 3.2: The Building-blocks of Software Characteristics

3.1.2 Integrating Malardalen Benchmarks

To incorporate Malardlen benchmarks to the automated test engine Figure 3.2 shows the needed building-blocks. The gray boxes that are shown in Figure 3.2 are the topic of this subsection.

The automated test engine uses the benchmarks in Tables 2.1, 2.2, and 2.3 for comparing a sheer number of different software characteristics that are running in different hardware architectures.

We want to see dynamic programming, migration of real-time tasks, and memory management in a lower level of abstractions. The lower level of abstraction is much more overwhelming to analyze. We can bypass this problem with tight integration to the automated test engine. What Malardalen benchmarks offer, a method of knowing what are the software characteristics in each specific benchmark. The automated test engine can know what software characteristics are running in what hardware architecture. If the automated test engine knows the name of the benchmark and the hardware configurations of the used hardware architecture, then the automated test engine can sort the overwhelming data through the regression algorithms.

For integrating the automated test engine to the Malardalen benchmarks, we simply need to develop a lookup table to allow the automated test engine to extract the relevant information to the regression algorithms.

Software Characteristic Lookup Table

The lookup table does not need to be sophisticated, the lookup table is a list type that is implemented in Python, and a string, which is the benchmark name, is used for indexing. In other words, we can use the name of the benchmark to map it to the software characteristics.

To simplify the lookup table, we categorize benchmarks in their software characteristics as shown in Table 3.1.

Table 3.1: Prototyping An Automated Engine Lookup Table

S	**L**	**N**	**A**	**B**	**R**	**U**	**F**	**Benchmark(s)**
	Yes							adpcm, nsichneu, statemate
	Yes		Yes					bs
	Yes	Yes	Yes					cnt, compress, fir, insertsort, ns, select
Yes	Yes							Cover, fibcall, Prime
Yes	Yes		Yes	Yes				Crc, fdct
Yes	Yes					Yes		duff
Yes	Yes	Yes	Yes	Yes				edn
Yes	Yes	Yes						expint, janne_complex, ud
Yes	Yes				Yes			fac
Yes	Yes	Yes	Yes				Yes	fft1, minver
Yes	Yes		Yes					jfdctint
	Yes			Yes				icdnum
Yes	Yes		Yes				Yes	lms, qurt
	Yes	Yes	Yes				Yes	ludcmp, qsort_exam
Yes	Yes	Yes	Yes					matmult
	Yes		Yes	Yes				ndes

By knowing the categories for these benchmarks, we can make each category to be a set and the benchmark name is the element of that set. For further simplification, suppose each set to be denoted to its corresponding software characteristics. For example, these benchmark fft1 and minver have the same software characteristics, which are: **S** for always a single path-program, **L** for containing loops, **N** for containing nested loops, **A** for using array and/or matrices, and **F** for using floating-point calculations. We can group these two benchmarks to the same software characteristic set, and let the set to be denoted as follows: **SLNAF**.

This method can simplify the implementation of the automated test engine. The indexing to each set would be through a loop, and the software characteristics would be gathered from the lookup table. The lookup table that would be used for the automated engine is shown in Table 3.2.

Table 3.2: The Automated Engine Lookup Table

Set:=	**{Element(s)}**
L:=	{adpcm, nsichneu, statemate}
LA:=	{bs}
LNA:=	{cnt, compress, fir, insertsort, ns, select}
SL:=	{Cover, fibcall, Prime}
SLAB:=	{fdct}
SLU:=	{duff}
SLNAB:=	{edn}
SLN:=	{expint, ud}
SLR:=	{fac}
SLNAF:=	{fft1, minver}
SLA:=	{jfdctint}
LB:=	{icdnum}
SLAF:=	{qurt}
LNAF:=	{ludcmp, qsort_exam}
SLNA:=	{matmult}
LAB:=	{ndes}

3.1.3 Integrating The Regression Algorithms

After execution times are obtained in our simulation environment, we resort to regression algorithm to capture the relationship between the execution times and hardware config-

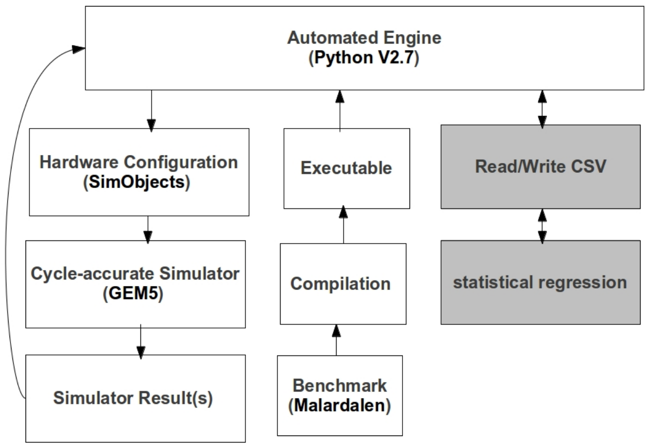

Figure 3.3: The Building-blocks of Regression Algorithms

urations. We integrate regression algorithms into our automated test engine, as the gray boxes are shown in Figure 3.3.

Robust Regression Algorithm

The robust regression algorithm approach is a solution to identify the relationship between the execution time and different hardware architecture configurations. The template equation of execution time estimation $\mathbf{T(M,I,C,F,B,S_2,A,\ S_1)}$ is the following:

$$\begin{aligned} T(M,I,C,F,B,S_2,A,S_1) = c_0 \cdot M + c_1 \cdot I + c_2 \cdot C \\ +c_3 \cdot F + c_4 \cdot B + c_5 \cdot S_2 + c_6 \cdot A + c_7 \cdot S_1 \end{aligned}$$

These are the input parameters:

M: Memory access

I: Number of instructions that the executable has

C: Number of cores in the system

F: The frequency of the system including the processor speed

B: Number of slots in TDMA bus

$\mathbf{S_2}$: The size of shared L2 cache

A: L2 Associativity

$\mathbf{S_1}$: The size of L1 cache

The purpose of robust regression algorithm is to find the $c_0, \cdots, c_7$ coefficients of the execution time estimation $\mathbf{T(M,I,C,F,B,S_2,A,\ S_1)}$. These $c_0, \cdots, c_7$ coefficients capture the relationship and thus develop an efficient way to rapidly estimate the program execution time under different hardware architecture configurations.

To improve the estimation accuracy, we classify programs with software characteristics to be different categories and develop the estimation models for each category separately. Table 3.2 shows these software characteristic categories.

The robust regression algorithms are integrated in the automated test engine because the robust regression algorithms can be used to detect outliers. The robust regression provides resistant and stable results in the presence of outliers, and the used tool to analyze the sparsity and consistency is $l_1 - norm$ regularized regression [15, 17]. The equation of robust regression problem with the uncertainty set for the $l_1 - norm$ regularized regression problem is the following [15]:

$$\underset{\beta \in R^m}{min} \{\|y - X\beta\|_2 + \sum_{i=1}^{m} c_i |\beta_i|\}$$

Where $\mathbf{y}$ is the independent variable $\mathbf{T(M,I,C,F,B,S_2,A,\ S_1)}$, or the set of admissible disturbances of the observed matrix $\mathbf{X}$, which is the dependent variables. The i_{th} is the iterator, β is the estimator. The estimator β_i is a fixed constant for each iteration runtime. $\mathbf{m}$ is the number of the conducted experiments, and $\mathbf{c_i}$ is the sought-out coefficients.

Robust estimators should be resistant to a certain degree of data contamination [23]. An acceptable robust estimator needs the robust regression algorithms to iterate more than once to minimize the error [15, 44]. The iterations are performed through a programming language, in this research we are using Python. There is a function in statsmodels library called RM() [4], which uses the $l_1 - norm$ regularized regression.

Statsmodels library is a Python module that is used for the following: estimating several different statistical models, conducting statistical tests, and statistical data exploration. The code for the regression algorithms is shown in appendix F with further explanations.

Simplified Example to Illustrate The Robust Regression Approach

The following example is simplified to show the reader the use of robust regression. Suppose we want to understand the relationship between execution time T and CPU frequency F, where the execution time T is the dependent variable and CPU frequency F is the independent variable. The other hardware and software configurations are constant for the sake of simplicity. Therefore, the execution time slop is $T = \hat{\beta}_1 + \hat{\beta}_2 F$.

We have conducted four experiments as Table 3.3 shows where the number of experiments is an arbitrary number.

Table 3.3: Simplified Example to Illustrate The Robust Regression Approach

Experiment	F_i	T_i	$\hat{T}_i$	$\tilde{T}_i$
1	**4 GHz**	0.696	1.464	-0.841
2	**3 GHz**	2.007	2.2	1.622
3	**2 GHz**	4.285	2.935	6.985
4	**1 GHz**	4.631	3.670	6.553

$\hat{T}_i$ is for each points had been moved vertically further from the regression line of $\hat{\beta}$. $\hat{\beta}$ is the estimator for the specific iteration, which in fact this example is dedicated to explain. Also, $\tilde{T}$ is the fitted value $\hat{T}_i$ summed with the residual $r_i = T_i - \hat{T}_i$.

We need to assume that the robust regression has iterated upto $N - 2$, and we are going to find the end result of the robust regression, which is $N-1$ of N iterations. Hence, N is an arbitrary number for how many iterations to minimize the estimation errors of the robust regression.

Because we are in the $N-2$ iteration, the estimator is given as follows $\hat{\beta}^{N-2} = \begin{pmatrix} 0.729 \\ 0.735 \end{pmatrix}$ by using the Python code in appendix F. The vertical distance from the line to each observation in the original sample is its residual $r_i = T_i - \hat{T}_i = T_i - (\hat{\beta}_1 + \hat{\beta}_2 F)$.

Since the $N-2$ iteration responses have simply been moved further from the estimation error given by $\hat{\beta}^{N-2} = \begin{pmatrix} 0.729 \\ 0.735 \end{pmatrix}$. By using the sample $(F_i, \tilde{T}_i)$ of the $N-2$ iteration, we can have the $N-1$ estimator, which is $\hat{\beta}^{N-1} = \begin{pmatrix} -1.962 \\ 1.888 \end{pmatrix}$. This iteration has fewer errors, the verification method is discussed in the next chapter.

3.1.4 Implementation of The Automated Test Engine

The simulator, software characteristic lookup table, and regression algorithms have been enhanced in order to make all of them able to share resources. To this point, all of these building-blocks cannot communicate to each other, so we need to implement the automated test engine, which allows them to exchange.

Figure 3.4 shows the needed building-blocks for connecting the simulator, benchmark lookup table, and the regression algorithms. The gray boxes that are shown in Figure 3.3 are incorporated into the source code of the automated test engine. The source code of the automated test engine is shown in appendix G with further implementation details.

3.2 Summary

In this chapter, we have enhanced and used GEM5, Malardlen benchmarks, and regression algorithms to cover the scope of the research. We started with an example then

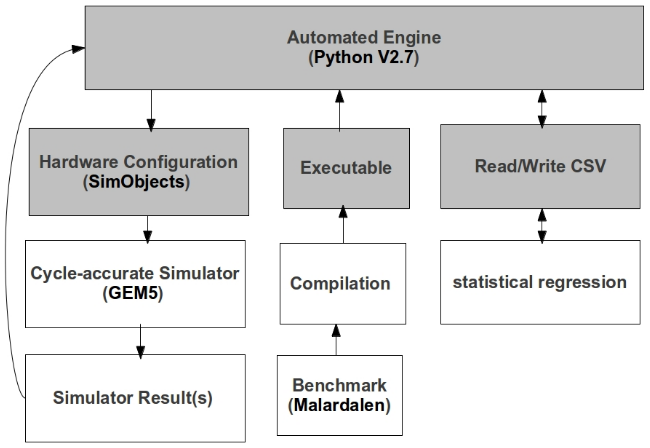

Figure 3.4: The Building-blocks of The Developed Automated Test Engine

we adjusted it to be compatible with other research building-blocks. After we have implemented the missing building-blocks of the automated test engine, we were able to integrate all of them into the automated test engine.

The automated test engine uses regression algorithms to derive the coefficients of the execution time estimation algorithms. The regression algorithms use the simulator output, but we want to categorize each benchmark in their software characteristics in order to increase the accuracy of execution time estimation. The method that we developed is a lookup table for the automated test engine to sort out the used benchmark in regards to their software characteristics.

CHAPTER 4

Experiments and Result Analysis

In chapter 4, we discuss the experiments we conducted and present the analytical formula we obtained through regression algorithm for estimating execution time of program in each category. We then present our validation efforts and results.

4.1 Using The Automated Test Engine

This section utilizes the automated test engine towards the research and shows the execution time estimation algorithms that are derived from the automated test engine. The implementation of the automated test engine has been discussed in chapter 3 and the source code is shown in the appendices.

4.1.1 Experiments

The needed parameters for running the automated test engine are already defined in the automated engine source code, so the only missing step is running the automated test engine. The command that runs the automated engine is shown below.

```
python run_automated_test_engine.py
```

Listing 4.1: Running The Automated Engine

This command is simple, but it is capable of preforming 357, 120 experiments, sorting their results, and deriving the regression algorithm for each set. The only element that is needed is for the user to wait for the automated test engine to finish and for the execution time estimation algorithms to be provided. The automated test engine took a month for conducting the 357, 120 experiments, then a week for the execution time estimation algorithms to be provided from the regression algorithms.

The automated test engine keeps all the data processing phases, so the raw data, the extracted data, and the sorted software characteristic sets are stored for debugging and backup purposes. These files will not be deleted by the automated test engine. For example, the conducted $357,120$ experiments use $50Gbytes$ from the hard disk.

4.1.2 Experiment Results

After the automated test engine finishes, one of its outputs is a file that contains the coefficients for each software characteristic set. These coefficients are used in the execution time estimation algorithms as shown in the next page.

From the execution time estimation algorithms, we can see that each equation corresponds to a specific software characteristic set, such as **L**, **LNA**, and **LNAF**. Each equation estimates the execution time for different hardware parameters which are represented in the equation as variables, such as **M**, **A**, and **B**. Each letter has already been defined, but they are mentioned here again to help the reader. These are the input parameters:

M: Memory access

I: Number of instructions that the program has

C: Number of cores in the system

F: The frequency of the system including the processor speed

B: Number of slots in TDMA bus

$\mathbf{S_2}$**:** The size of shared L2 cache

A: L2 Associativity

$\mathbf{S_1}$**:** The size of L1 cache

$$T = \begin{cases}
(8.4 \cdot 10^{-10}) \cdot M + (2.2 \cdot 10^{-10}) \cdot I + (-2.8 \cdot 10^{-5}) \cdot C + (-6.45 \cdot 10^{-5}) \cdot F + (-2.5 \cdot 10^{-7}) \cdot B + (-2.3 \cdot 10^{-6}) \cdot S_2 + (-9.7 \cdot 10^{-7}) \cdot A + (-2.9 \cdot 10^{-7}) \cdot S_1 & \textbf{L} \\
(2.19 \cdot 10^{-10}) \cdot M + (-7.52 \cdot 10^{-8}) \cdot I + (0.3 \cdot 10^{-3}) \cdot C + (-1.63 \cdot 10^{-5}) \cdot F + (1.2 \cdot 10^{-7}) \cdot B + (-5.15 \cdot 10^{-7}) \cdot S_2 + (1.1 \cdot 10^{-8}) \cdot A + (1.48 \cdot 10^{-8}) \cdot S_1 & \textbf{LA} \\
(7.67 \cdot 10^{-10}) \cdot M + (1.46 \cdot 10^{-8}) \cdot I + (-1.3 \cdot 10^{-3}) \cdot C + (-6.31 \cdot 10^{-5}) \cdot F + (2.9 \cdot 10^{-7}) \cdot B + (-1.8 \cdot 10^{-6}) \cdot S_2 + (5.7 \cdot 10^{-8}) \cdot A + (-2.04 \cdot 10^{-8}) \cdot S_1 & \textbf{LAB} \\
(2.2 \cdot 10^{-10}) \cdot M + (-4.3 \cdot 10^{-8}) \cdot I + (0.2 \cdot 10^{-3}) \cdot C + (-1.63 \cdot 10^{-5}) \cdot F + (1.6 \cdot 10^{-7}) \cdot B + (-5.1 \cdot 10^{-7}) \cdot S_2 + (1.5 \cdot 10^{-8}) \cdot A + (-3.7 \cdot 10^{-8}) \cdot S_1 & \textbf{LB} \\
(2.8 \cdot 10^{-10}) \cdot M + (3.07 \cdot 10^{-10}) \cdot I + (-7.6 \cdot 10^{-6}) \cdot C + (-2.1 \cdot 10^{-5}) \cdot F + (1.6 \cdot 10^{-7}) \cdot B + (-6.6 \cdot 10^{-7}) \cdot S_2 + (1.1 \cdot 10^{-8}) \cdot A + (-4.9 \cdot 10^{-8}) \cdot S_1 & \textbf{LNA} \\
(2.5 \cdot 10^{-10}) \cdot M + (1.9 \cdot 10^{-10}) \cdot I + (-4.5 \cdot 10^{-6}) \cdot C + (-1.9 \cdot 10^{-5}) \cdot F + (1.45 \cdot 10^{-7}) \cdot B + (-5.9 \cdot 10^{-7}) \cdot S_2 + (1.43 \cdot 10^{-8}) \cdot A + (-7.7 \cdot 10^{-8}) \cdot S_1 & \textbf{LNAF} \\
(2.6 \cdot 10^{-10}) \cdot M + (-3.6 \cdot 10^{-8}) \cdot I + (0.3 \cdot 10^{-3}) \cdot C + (-1.96 \cdot 10^{-5}) \cdot F + (1.7 \cdot 10^{-7}) \cdot B + (-6.15 \cdot 10^{-7}) \cdot S_2 + (2.2 \cdot 10^{-8}) \cdot A + (-7.04 \cdot 10^{-8}) \cdot S_1 & \textbf{SLA} \\
(3.5 \cdot 10^{-10}) \cdot M + (3.2 \cdot 10^{-10}) \cdot I + (-1.1 \cdot 10^{-5}) \cdot C + (-2.67 \cdot 10^{-5}) \cdot F + (1.59 \cdot 10^{-7}) \cdot B + (-8.36 \cdot 10^{-7}) \cdot S_2 + (2.44 \cdot 10^{-8}) \cdot A + (-4.97 \cdot 10^{-8}) \cdot S_1 & \textbf{SLAB} \\
(1.36 \cdot 10^{-9}) \cdot M + (2.05 \cdot 10^{-10}) \cdot I + (-5.9 \cdot 10^{-5}) \cdot C + (-0.1 \cdot 10^{-3}) \cdot F + (2.9 \cdot 10^{-7}) \cdot B + (-3.2 \cdot 10^{-6}) \cdot S_2 + (9.46 \cdot 10^{-8}) \cdot A + (-7.28 \cdot 10^{-8}) \cdot S_1 & \textbf{SLAF} \\
(2.47 \cdot 10^{-10}) \cdot M + (1.6 \cdot 10^{-10}) \cdot I + (-4.4 \cdot 10^{-6}) \cdot C + (-1.8 \cdot 10^{-5}) \cdot F + (-1.3 \cdot 10^{-7}) \cdot B + (-5.8 \cdot 10^{-7}) \cdot S_2 + (1.59 \cdot 10^{-8}) \cdot A + (-4.3 \cdot 10^{-8}) \cdot S_1 & \textbf{SL} \\
(1.39 \cdot 10^{-9}) \cdot M + (-2.6 \cdot 10^{-7}) \cdot I + (89.8 \cdot 10^{-3}) \cdot C + (-0.1 \cdot 10^{-3}) \cdot F + (2.04 \cdot 10^{-7}) \cdot B + (-3.27 \cdot 10^{-6}) \cdot S_2 + (6.2 \cdot 10^{-8}) \cdot A + (-1.5 \cdot 10^{-6}) \cdot S_1 & \textbf{SLNA} \\
(8.1 \cdot 10^{-10}) \cdot M + (5.1 \cdot 10^{-8}) \cdot I + (-5.7 \cdot 10^{-3}) \cdot C + (-6.7 \cdot 10^{-5}) \cdot F + (2.3 \cdot 10^{-7}) \cdot B + (-1.9 \cdot 10^{-6}) \cdot S_2 + (8.4 \cdot 10^{-8}) \cdot A + (-4.7 \cdot 10^{-8}) \cdot S_1 & \textbf{SLNAB} \\
(2.49 \cdot 10^{-10}) \cdot M + (3.58 \cdot 10^{-10}) \cdot I + (-5.5 \cdot 10^{-6}) \cdot C + (-1.89 \cdot 10^{-5}) \cdot F + (1.68 \cdot 10^{-7}) \cdot B + (-5.8 \cdot 10^{-7}) \cdot S_2 + (1.85 \cdot 10^{-8}) \cdot A + (-7.8 \cdot 10^{-8}) \cdot S_1 & \textbf{SLNAF} \\
(2.3 \cdot 10^{-10}) \cdot M + (2.87 \cdot 10^{-10}) \cdot I + (-4.84 \cdot 10^{-6}) \cdot C + (-1.75 \cdot 10^{-5}) \cdot F + (1.68 \cdot 10^{-7}) \cdot B + (-5.47 \cdot 10^{-7}) \cdot S_2 + (1.6 \cdot 10^{-8}) \cdot A + (-4 \cdot 10^{-8}) \cdot S_1 & \textbf{SLN} \\
(2.19 \cdot 10^{-10}) \cdot M + (-4.08 \cdot 10^{-8}) \cdot I + (0.2 \cdot 10^{-3}) \cdot C + (-1.6 \cdot 10^{-5}) \cdot F + (1.56 \cdot 10^{-7}) \cdot B + (-5.1 \cdot 10^{-7}) \cdot S_2 + (1.67 \cdot 10^{-8}) \cdot A + (-5.6 \cdot 10^{-8}) \cdot S_1 & \textbf{SLR} \\
(2.2 \cdot 10^{-10}) \cdot M + (6.94 \cdot 10^{-8}) \cdot I + (-0.4 \cdot 10^{-3}) \cdot C + (-1.68 \cdot 10^{-5}) \cdot F + (1.7 \cdot 10^{-7}) \cdot B + (-5.2 \cdot 10^{-7}) \cdot S_2 + (1.26 \cdot 10^{-8}) \cdot A + (-4.08 \cdot 10^{-8}) \cdot S_1 & \textbf{SLU}
\end{cases}$$

The left-side of the equations is the dependent variable, which is the execution time. The execution time estimation algorithms that are shown in the previous page use T as the execution time. However, there is a better mathematical notation to represent the execution time, as shown below:

$$T \equiv T(M, I, C, F, B, S_2, A, S_1)$$

The reason for not showing the better mathematical notation with the execution time estimation algorithms is because the algorithms we developed are long, so if we include this $T(M, I, C, F, B, S_2, A, S_1)$, then part of the algorithms would not show on the page. Therefore, there was a need to remove part of the algorithm notation and make sure all the equation terms are shown. We are going to use $T(M, I, C, F, B, S_2, A, S_1)$ instead of T.

4.1.3 Valid Experiment Results

Unfortunately, the used validation method for execution time estimation does not support all the equations because the used validation method has a restriction on the sample number of benchmarks. Table 3.2, which is the lookup table of the automated test engine, shows that **LA**, **SLU**, **SLNAB**, **SLR**, **SLA**, **LB**, **SLNA**, **SLAB**, and **LAB** sets have just one benchmark in their set.

We need at least two benchmarks that have similar software characteristics to do the used validation method for the derived execution time estimation. Because these benchmarks are adopted from Malardalen benchmarks, this is out of our control. Therefore, the used validation method uses software characteristic sets that have more than one benchmark, which are the following software characteristic sets: **L**, **LNA**, **LNAF**, **SL**, **SLNAF**. The selected software characteristic equations for validations are shown in the next page.

$$T = \begin{cases} (8.4 \cdot 10^{-10}) \cdot M + (2.2 \cdot 10^{-10}) \cdot I + (-2.8 \cdot 10^{-5}) \cdot C + (-6.45 \cdot 10^{-5}) \cdot F + (-2.5 \cdot 10^{-7}) \cdot B + (-2.3 \cdot 10^{-6}) \cdot S_2 + (-9.7 \cdot 10^{-7}) \cdot A + (-2.9 \cdot 10^{-7}) \cdot S_1 & \textbf{L} \\ (2.8 \cdot 10^{-10}) \cdot M + (3.07 \cdot 10^{-10}) \cdot I + (-7.6 \cdot 10^{-6}) \cdot C + (-2.1 \cdot 10^{-5}) \cdot F + (1.6 \cdot 10^{-7}) \cdot B + (-6.6 \cdot 10^{-7}) \cdot S_2 + (1.1 \cdot 10^{-8}) \cdot A + (-4.9 \cdot 10^{-8}) \cdot S_1 & \textbf{LNA} \\ (2.5 \cdot 10^{-10}) \cdot M + (1.9 \cdot 10^{-10}) \cdot I + (-4.5 \cdot 10^{-6}) \cdot C + (-1.9 \cdot 10^{-5}) \cdot F + (1.45 \cdot 10^{-7}) \cdot B + (-5.9 \cdot 10^{-7}) \cdot S_2 + (1.43 \cdot 10^{-8}) \cdot A + (-7.7 \cdot 10^{-8}) \cdot S_1 & \textbf{LNAF} \\ (2.47 \cdot 10^{-10}) \cdot M + (1.6 \cdot 10^{-10}) \cdot I + (-4.4 \cdot 10^{-6}) \cdot C + (-1.8 \cdot 10^{-5}) \cdot F + (-1.3 \cdot 10^{-7}) \cdot B + (-5.8 \cdot 10^{-7}) \cdot S_2 + (1.59 \cdot 10^{-8}) \cdot A + (-4.3 \cdot 10^{-8}) \cdot S_1 & \textbf{SL} \\ (2.49 \cdot 10^{-10}) \cdot M + (3.58 \cdot 10^{-10}) \cdot I + (-5.5 \cdot 10^{-6}) \cdot C + (-1.89 \cdot 10^{-5}) \cdot F + (1.68 \cdot 10^{-7}) \cdot B + (-5.8 \cdot 10^{-7}) \cdot S_2 + (1.85 \cdot 10^{-8}) \cdot A + (-7.8 \cdot 10^{-8}) \cdot S_1 & \textbf{SLNAF} \end{cases}$$

4.2 Accuracy Validation for Execution Time Estimation

In this section, we examine the accuracy of the execution time estimation algorithms, so we are plotting the fitted model for each software characteristic set. The x-axis of the plot is the estimated execution time, and the y-axis is the conducted execution time. The purpose of this plot is to show the accuracy and precision for each execution time estimation algorithm.

In addition to the error plot for each software characteristic set, we are also examining these aspects, which are the minimal percentage error, the maximal percentage error, the average (mean) percentage error, and the variance of each software characteristic set.

4.2.1 Accuracy Evaluation for L

The equation of **L** to estimate the execution time is shown as follows:

$$\begin{aligned} T(M, I, C, F, B, S_2, A, S_1) = (8.4 \cdot 10^{-10}) \cdot M + (2.2 \cdot 10^{-10}) \cdot I \\ +(-2.8 \cdot 10^{-5}) \cdot C + (-6.45 \cdot 10^{-5}) \cdot F + (-2.5 \cdot 10^{-7}) \cdot B \\ +(-2.3 \cdot 10^{-6}) \cdot S_2 + (-9.7 \cdot 10^{-7}) \cdot A + (-2.9 \cdot 10^{-7}) \cdot S_1 \end{aligned}$$

We used both training data and non-training data accuracy validation methods for the equation of **L**.

Accuracy Validation for The Training Data

The fitted model error plot is shown in Figure 4.1. The minimal percentage error, the maximal percentage error, the average (mean) percentage error, and the variance of each software characteristic set are shown in Table 4.1.

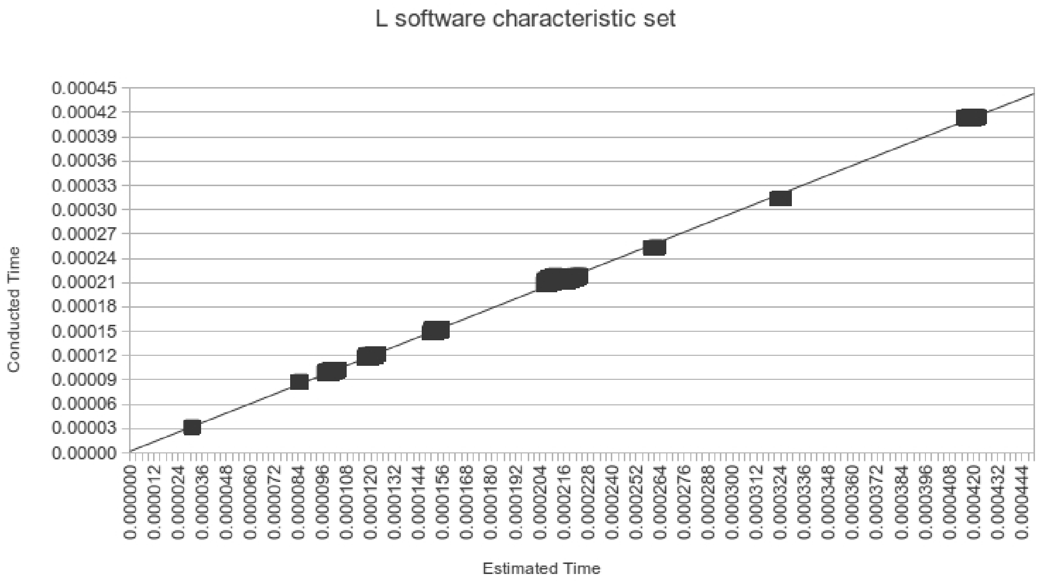

Figure 4.1: The Error Plot of L Set for The Training Data

Table 4.1: L Execution Time Estimation for The Training Data

Name	**Value**
Minimal Estimated Error	0.008%
Maximal Estimated Error	3.68%
Average Estimated Error	0.93%
Variance Between Errors	$1.1 \cdot 10^{-4}$

Accuracy Validation for The Non-training Data

We conducted further 2400 experiments that were not used in the training data. We have introduced these two new configurations $\mathbf{S_2}= 4MB$ and $\mathbf{A}= 2-way$. The fitted model error plot is shown in Figure 4.2. The minimal percentage error, the maximal percentage error, the average (mean) percentage error, and the variance of each software characteristic set are shown in Table 4.2.

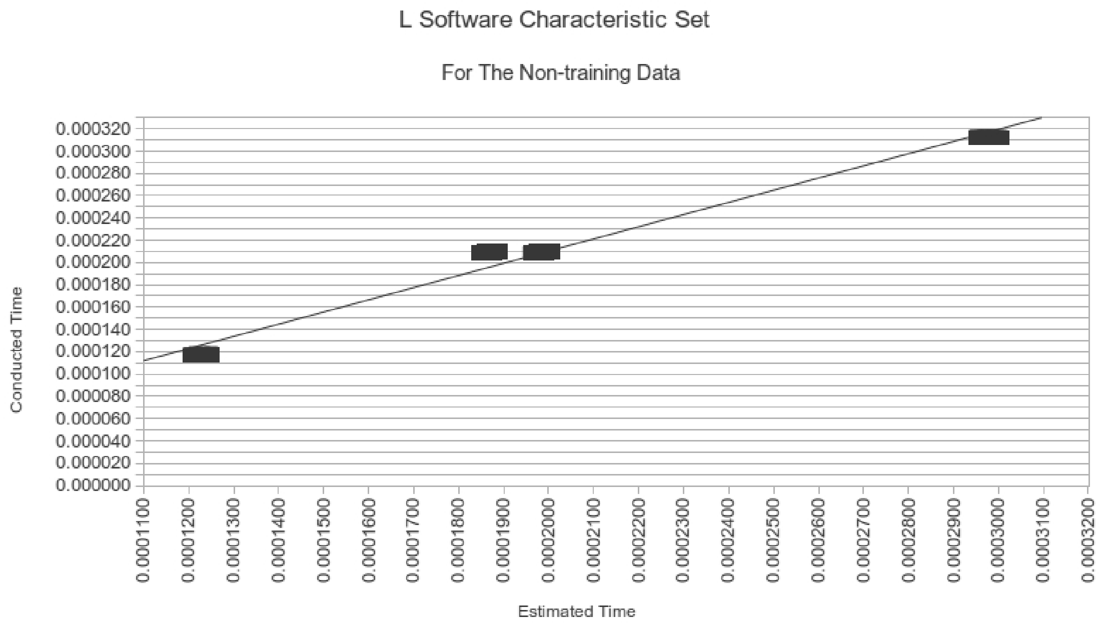

Figure 4.2: The Error Plot of L Set for The Non-training Data

Table 4.2: L Execution Time Estimation for The Non-training Data

Name	**Value**
Minimal Estimated Error	1.84%
Maximal Estimated Error	11.73%
Average Estimated Error	6.25%
Variance Between Errors	$7.16 \cdot 10^{-4}$

The below Table 4.3 shows five conducted experiments that are not from the training data.

Table 4.3: Non-training Data Error Evaluation of L

Experiment	Benchmark	C	F	B	Conducted Time	Measured Time
1	**nsichneu**	4	1.5	10	0.000154	0.0001554056
2	**adpcm**	5	2	4	0.000313	0.0003277014
3	**adpcm**	5	2	10	0.000314	0.0003283083
4	**statemate**	3	3.5	10	0.000032	3.13395822E-005
5	**nsichneu**	2	3	9	0.000089	8.45587339E-005

Note: the storage is fixed for all the experiments in Table 4.3, which are: $\mathbf{S_2}= 8MB$, $\mathbf{A}= 4-way$, and $\mathbf{S_1}=8KB$.

4.2.2 Accuracy Evaluation for LNA

The equation of **LNA** to estimate the execution time is shown as follows:

$$\begin{aligned}T(M, I, C, F, B, S_2, A, S_1) = (2.8 \cdot 10^{-10}) \cdot M + (3.07 \cdot 10^{-10}) \cdot I \\ +(-7.6 \cdot 10^{-6}) \cdot C + (-2.1 \cdot 10^{-5}) \cdot F + (1.6 \cdot 10^{-7}) \cdot B \\ +(-6.6 \cdot 10^{-7}) \cdot S_2 + (1.1 \cdot 10^{-8}) \cdot A + (-4.9 \cdot 10^{-8}) \cdot S_1\end{aligned}$$

We used both training data and non-training data accuracy validation methods for the equation of **LNA**.

Accuracy Validation for The Training Data

The fitted model error plot is shown in Figure 4.3. The minimal percentage error, the maximal percentage error, the average (mean) percentage error, and the variance of each software characteristic set are shown in Table 4.4.

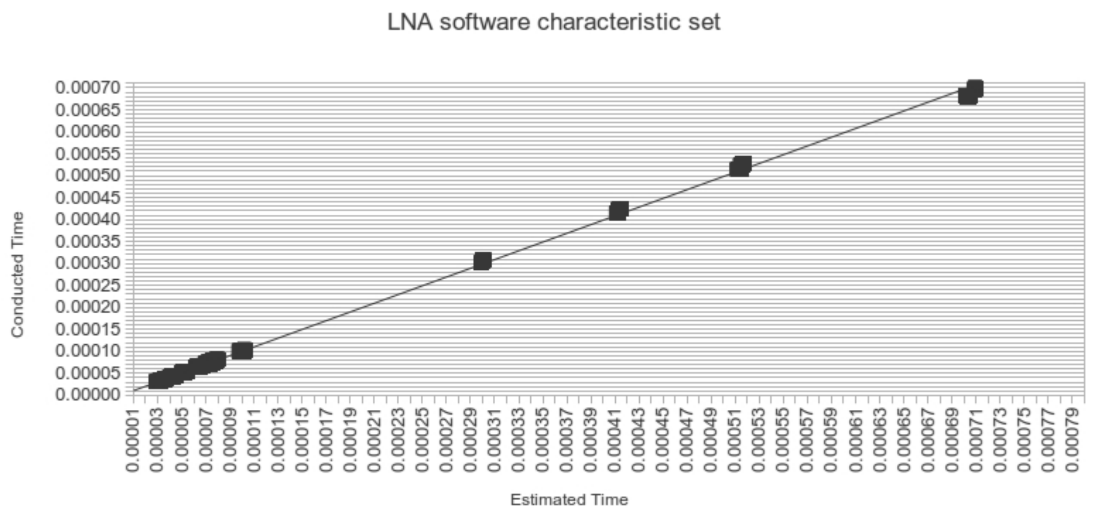

Figure 4.3: The Error Plot of LNA Set

Table 4.4: LNA Execution Time Estimation for The Training Data

Name	**Value**
Minimal Estimated Error	0.0%
Maximal Estimated Error	3.74%
Average Estimated Error	1.89%
Variance Between Errors	$1.207 \cdot 10^{-4}$

Accuracy Validation for The Non-training Data

We conducted further 2400 experiments that were not used in the training data. We have introduced these two new configurations $\mathbf{S_2}= 4MB$ and $\mathbf{A}= 2-way$. The fitted model error plot is shown in Figure 4.4. The minimal percentage error, the maximal percentage error, the average (mean) percentage error, and the variance of each software characteristic set are shown in Table 4.5.

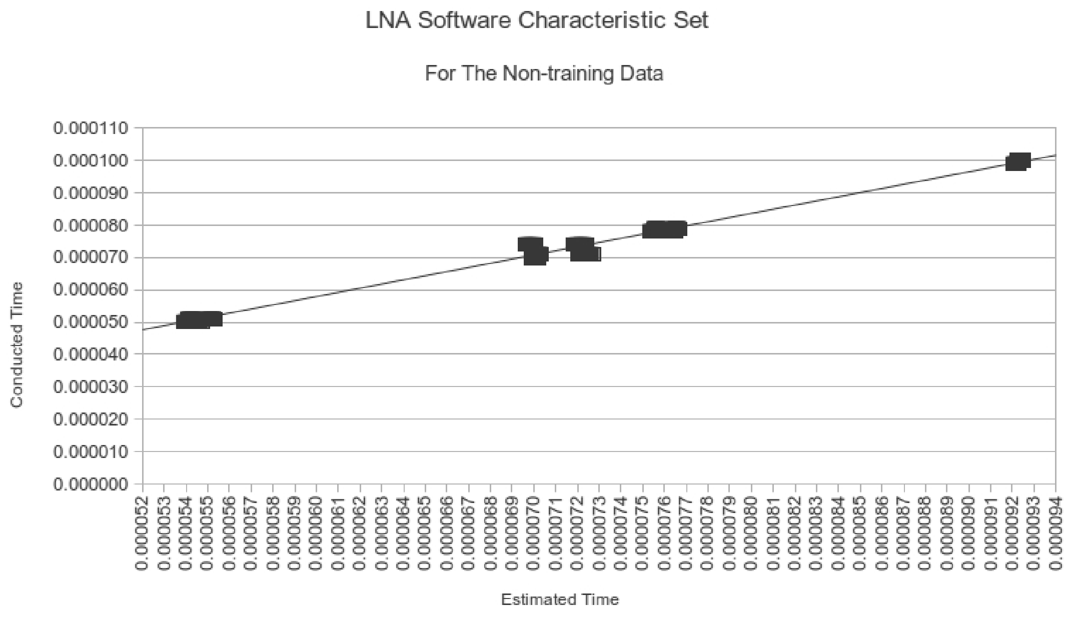

Figure 4.4: The Error Plot of LNA Set for The Non-training Data

Table 4.5: LNA Execution Time Estimation for The Non-training Data

Name	Value
Minimal Estimated Error	2.03%
Maximal Estimated Error	7.84%
Average Estimated Error	4.22%
Variance Between Errors	$6.56 \cdot 10^{-4}$

The below Table 4.6 shows five conducted experiments that are not from the training data.

Table 4.6: Non-training Data Error Evaluation of LNA

Experiment	**Benchmark**	**C**	**F**	**B**	**Conducted Time**	**Measured Time**
1	**fir**	7	1.5	10	0.0007	0.0007097631
2	**insertsort**	3	1	1	0.000071	7.23161185E-005
3	**ns**	6	1	8	0.000102	0.000099184
4	**cnt**	5	3	2	0.000033	0.000031853
5	**cnt**	3	1	1	0.000079	7.62453377E-005

Note: the storage is fixed for all the experiments in Table 4.6, which are: $\mathbf{S_2}= 4MB$, $\mathbf{A}= 4-way$, and $\mathbf{S_1}=8KB$.

4.2.3 Accuracy Evaluation for LNAF

The equation of **LNAF** to estimate the execution time is shown as follows:

$$\begin{aligned} T(M, I, C, F, B, S_2, A, S_1) &= (2.5 \cdot 10^{-10}) \cdot M + (1.9 \cdot 10^{-10}) \cdot I \\ &+(-4.5 \cdot 10^{-6}) \cdot C + (-1.9 \cdot 10^{-5}) \cdot F + (1.45 \cdot 10^{-7}) \cdot B \\ &+(-5.9 \cdot 10^{-7}) \cdot S_2 + (1.43 \cdot 10^{-8}) \cdot A + (-7.7 \cdot 10^{-8}) \cdot S_1 \end{aligned}$$

We used both training data and non-training data accuracy validation methods for the equation of **LNAF**.

Accuracy Validation for The Training Data

The fitted model error plot is shown in Figure 4.5. The minimal percentage error, the maximal percentage error, the average (mean) percentage error, and the variance of each software characteristic set are shown in Table 4.7.

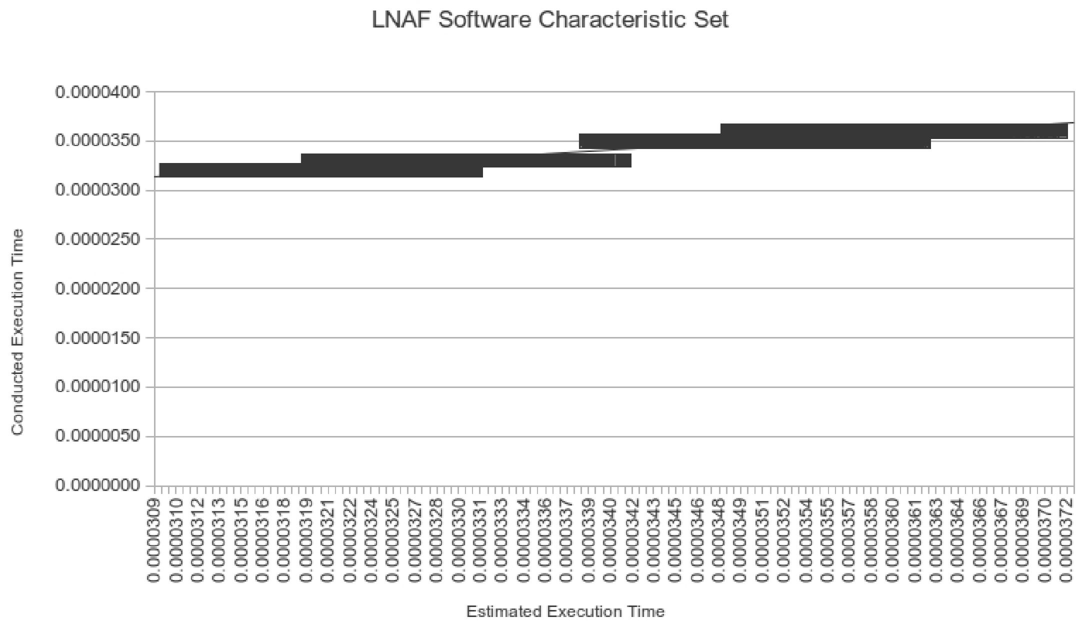

Figure 4.5: The Error Plot of LNAF Set

Table 4.7: LNAF Execution Time Estimation for The Training Data

Name	**Value**
Minimal Estimated Error	0.0%
Maximal Estimated Error	3.33%
Average Estimated Error	1.56%
Variance Between Errors	$8.76266 \cdot 10^{-5}$

Accuracy Validation for The Non-training Data

We conducted further 2400 experiments that were not used in the training data. We have introduced these two new configurations $\mathbf{S_2}= 4MB$ and $\mathbf{A}= 2-way$. The fitted model error plot is shown in Figure 4.6. The minimal percentage error, the maximal percentage error, the average (mean) percentage error, and the variance of each software characteristic set are shown in Table 4.8.

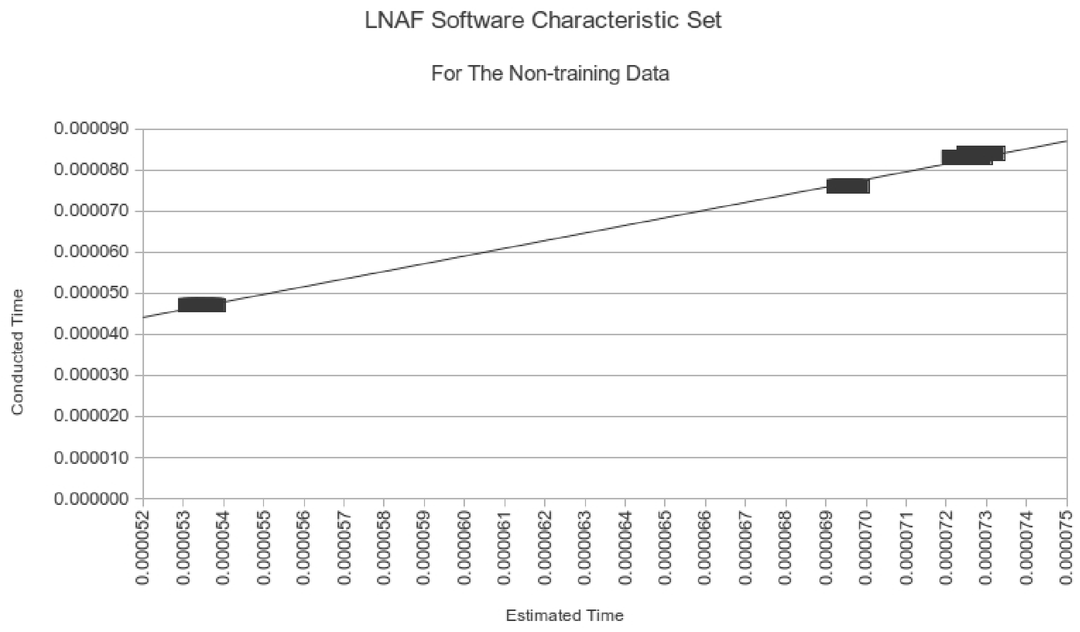

Figure 4.6: The Error Plot of LNAF Set for The Non-training Data

Table 4.8: LNAF Execution Time Estimation for The Non-training Data

Name	**Value**
Minimal Estimated Error	8.09%
Maximal Estimated Error	14.61%
Average Estimated Error	11.50%
Variance Between Errors	$5.52 \cdot 10^{-4}$

The below Table 4.9 shows five conducted experiments that are not from the training data.

Table 4.9: Non-training Data Error Evaluation of LNAF

Experiment	Benchmark	C	F	B	Conducted Time	Measured Time
1	**qsort-exam**	6	1.5	1	0.000054	5.82381438E-005
2	**qsort-exam**	5	3	3	0.000032	3.05091654E-005
3	**qsort-exam**	2	3	9	0.000032	3.26237378E-005
4	**ludcmp**	2	1.5	5	0.000059	6.22609006E-005
5	**qsort-exam**	6	2.5	4	0.000037	3.96525438E-005

Note: the storage is fixed for all the experiments in Table 4.9, which are: $\mathbf{S_2}= 16MB$, $\mathbf{A}= 8-way$, and $\mathbf{S_1}=16KB$.

4.2.4 Accuracy Evaluation for SL

The equation of **SL** to estimate the execution time is shown as follows:

$$\begin{aligned} T(M,I,C,F,B,S_2,A,S_1) &= (2.47\cdot 10^{-10})\cdot M + (1.6\cdot 10^{-10})\cdot I \\ &+(-4.4\cdot 10^{-6})\cdot C + (-1.8\cdot 10^{-5})\cdot F + (-1.3\cdot 10^{-7})\cdot B \\ &+(-5.8\cdot 10^{-7})\cdot S_2 + (1.59\cdot 10^{-8})\cdot A + (-4.3\cdot 10^{-8})\cdot S_1 \end{aligned}$$

We used both training data and non-training data accuracy validation methods for the equation of **SL**.

Accuracy Validation for The Training Data

The fitted model error plot is shown in Figure 4.7. The minimal percentage error, the maximal percentage error, the average (mean) percentage error, and the variance of each software characteristic set are shown in Table 4.10.

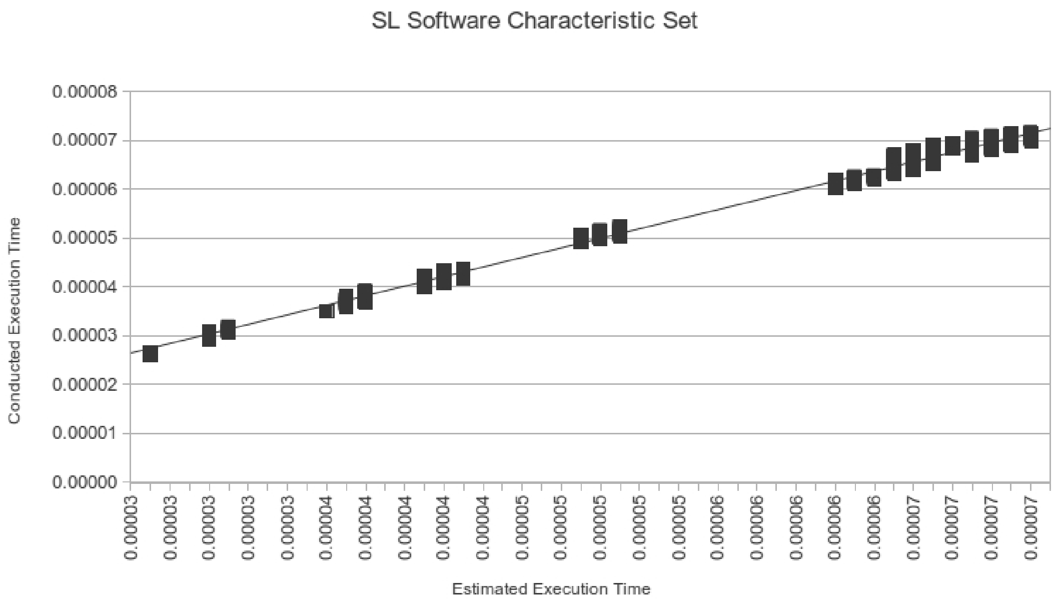

Figure 4.7: The Error Plot of SL Set

Table 4.10: SL Execution Time Estimation for The Training Data

Name	**Value**
Minimal Estimated Error	0.0%
Maximal Estimated Error	2.98%
Average Estimated Error	1.51%
Variance Between Errors	$7.2522824 \cdot 10^{-5}$

Accuracy Validation for The Non-training Data

We conducted further 2400 experiments that were not used in the training data. We have introduced these two new configurations $\mathbf{S_2}= 4MB$ and $\mathbf{A}= 2-way$. The fitted model error plot is shown in Figure 4.8. The minimal percentage error, the maximal percentage error, the average (mean) percentage error, and the variance of each software characteristic set are shown in Table 4.11.

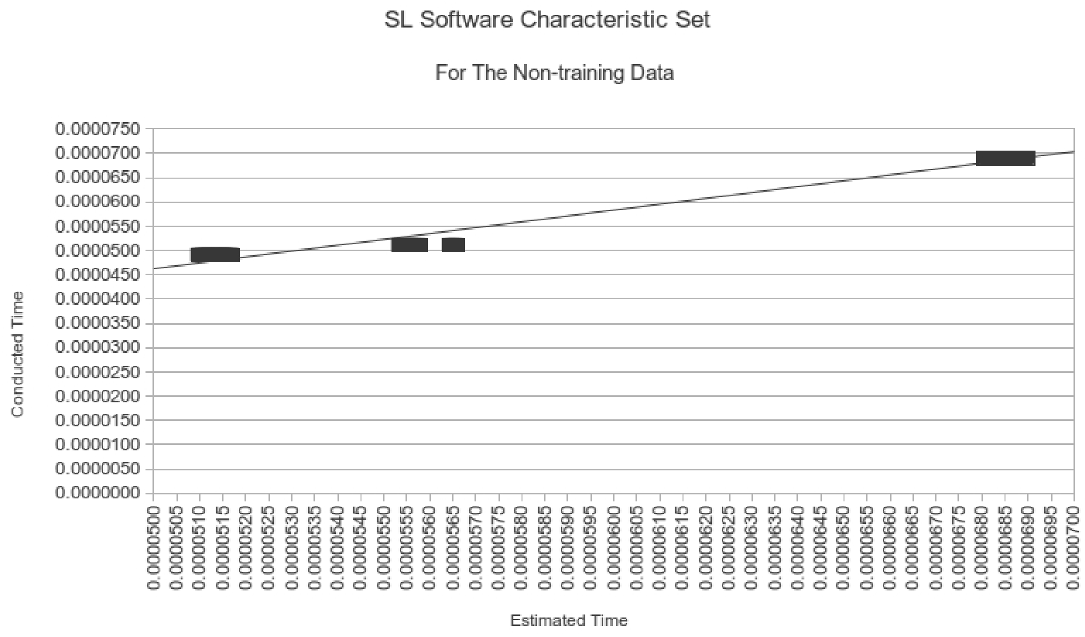

Figure 4.8: The Error Plot of SL Set for The Non-training Data

Table 4.11: SL Execution Time Estimation for The Non-training Data

Name	**Value**
Minimal Estimated Error	0.00%
Maximal Estimated Error	9.47%
Average Estimated Error	4.47%
Variance Between Errors	$1.21 \cdot 10^{-4}$

The below Table 4.12 shows five conducted experiments that are not from the training data.

Table 4.12: Non-training Data Error Evaluation of SL

Experiment	Benchmark	C	F	B	Conducted Time	Measured Time
1	**prime**	2	1.5	10	0.000067	6.48865942E-005
2	**cover**	4	2	9	0.00005	5.21151374E-005
3	**fibcall**	3	3	2	0.00003	3.25455584E-005
4	**prime**	5	3	5	0.000038	3.9819649E-005
5	**cover**	4	2.5	1	0.000041	4.20661374E-005

Note: the storage is fixed for all the experiments in table 4.12, which are: $\mathbf{S_2}$= $16MB$, **A**= $8-way$, and $\mathbf{S_1}$=$8KB$.

4.2.5 Accuracy Evaluation for SLNAF

The equation of **SLNAF** to estimate the execution time is shown as follows:

$$\begin{aligned} T(M,I,C,F,B,S_2,A,S_1) &= (2.49\cdot10^{-10})\cdot M + (3.58\cdot10^{-10})\cdot I \\ &+(-5.5\cdot10^{-6})\cdot C + (-1.89\cdot10^{-5})\cdot F + (1.68\cdot10^{-7})\cdot B \\ &+(-5.8\cdot10^{-7})\cdot S_2 + (1.85\cdot10^{-8})\cdot A + (-7.8\cdot10^{-8})\cdot S_1 \end{aligned}$$

We used both training data and non-training data accuracy validation methods for the equation of **SLNAF**.

Accuracy Validation for The Training Data

The fitted model error plot is shown in Figure 4.9. The minimal percentage error, the maximal percentage error, the average (mean) percentage error, and the variance of each software characteristic set are shown in Table 4.13.

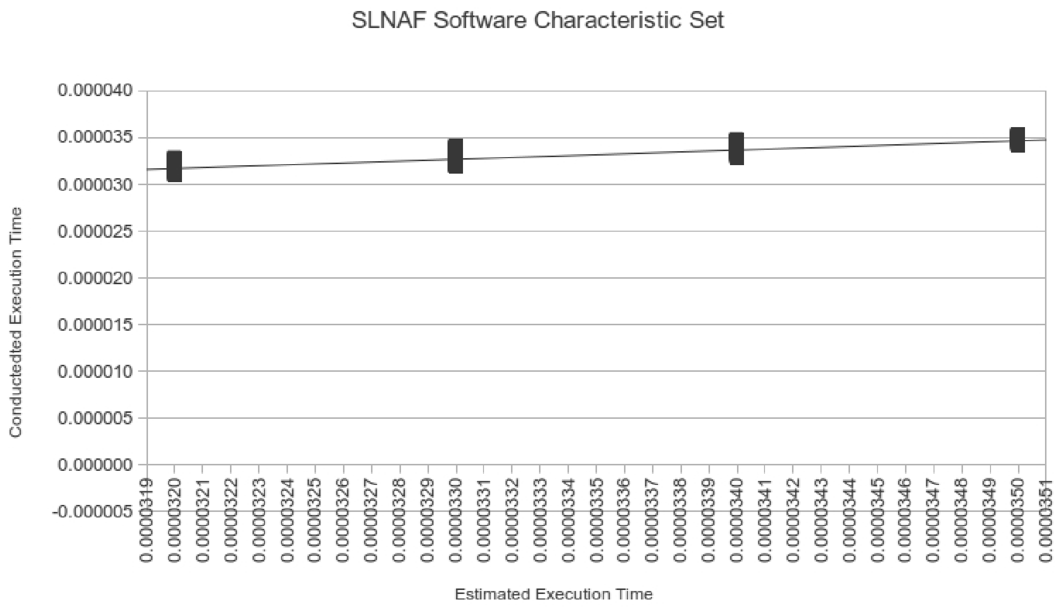

Figure 4.9: The Error Plot of SLNAF Set

Table 4.13: SLNAF Execution Time Estimation for The Training Data

Name	**Value**
Minimal Estimated Error	0.0%
Maximal Estimated Error	3.34%
Average Estimated Error	1.04%
Variance Between Errors	$8.60414 \cdot 10^{-5}$

Accuracy Validation for The Non-training Data

We conducted further 2400 experiments that were not used in the training data. We have introduced these two new configurations $\mathbf{S_2}= 4MB$ and $\mathbf{A}= 2-way$. The fitted model error plot is shown in Figure 4.10. The minimal percentage error, the maximal percentage error, the average (mean) percentage error, and the variance of each software characteristic set are shown in Table 4.14.

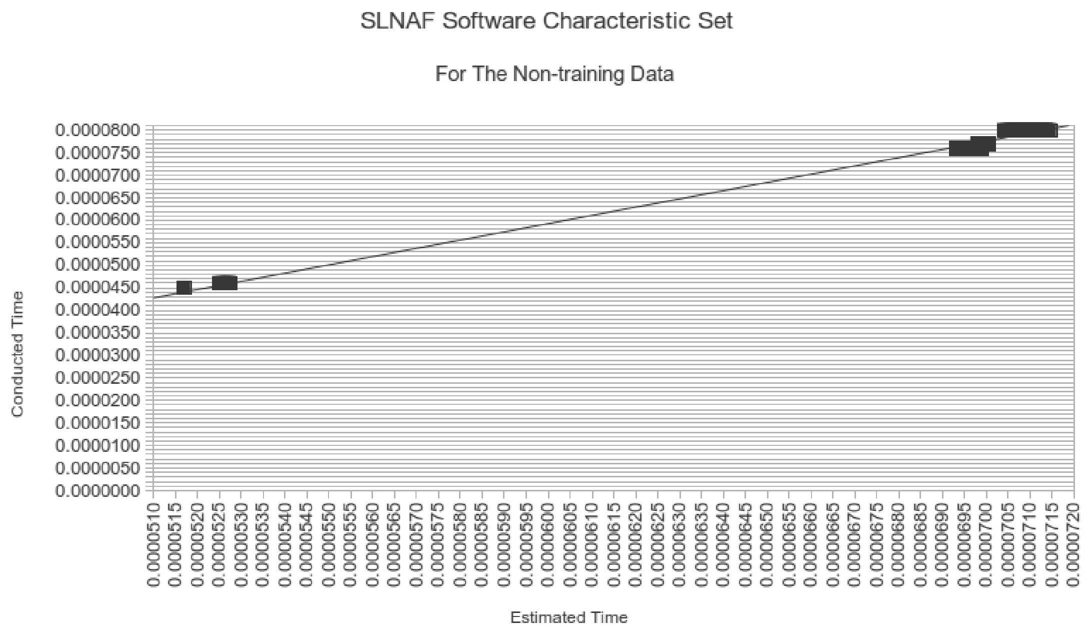

Figure 4.10: The Error Plot of SLNAF Set for The Non-training Data

Table 4.14: SLNAF Execution Time Estimation for The Non-training Data

Name	**Value**
Minimal Estimated Error	8.06%
Maximal Estimated Error	14.90%
Average Estimated Error	10.27%
Variance Between Errors	$3.15 \cdot 10^{-4}$

The below Table 4.15 shows five conducted experiments that are not from the training data.

Table 4.15: Non-training Data Error Evaluation of SLNAF

Experiment	**Benchmark**	**C**	**F**	**B**	**Conducted Time**	**Measured Time**
1	**minver**	6	1.5	10	0.00006	6.34795268E-005
2	**fft1**	6	3	7	0.000033	3.19581046E-005
3	**fft1**	5	1.5	9	0.000057	0.000060998
4	**fft1**	6	1.5	8	0.000057	6.04476998E-005
5	**fft1**	5	3	1	0.000033	3.13302216E-005

Note: the storage is fixed for all the experiments in Table 4.15, which are: $\mathbf{S_2}= 4MB$, $\mathbf{A}= 8-way$, and $\mathbf{S_1}=8KB$.

4.3 Execution Time with A Given Hardware Architecture

In this section, we are going to write a program and estimate the execution time of that program. In fact, the execution time estimation algorithms that we have developed are created to accomplish this objective. For the sake of simplicity, we are going to assume that we are given a hardware architecture and a program. Our task is to estimate the execution time.

Suppose the given hardware is 4 cores, as shown in Figure 4.11. The hardware parameters for the design of Figure 4.11 are as follows:

C: 4

F: $2GHz$

B: 2 slots

$\mathbf{S_2}$**:** $4MB$

A: 4-way associativity

$\mathbf{S_1}$**:** $8kB$

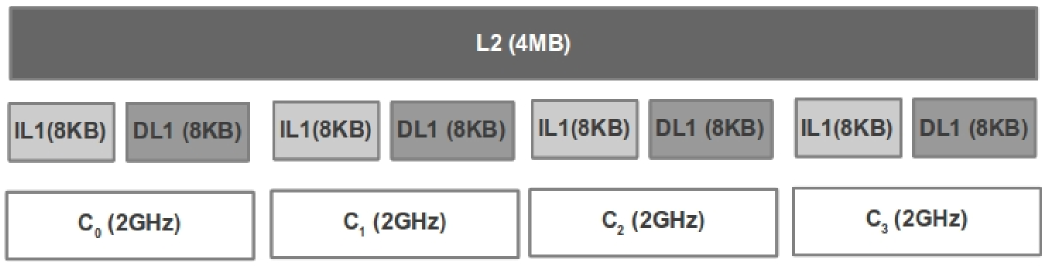

Figure 4.11: Given Hardware Architecture: 4 Cores System

The program that we are using calculates a factorial of a given number. Because we are just focusing on the software characteristics, we are not including IO delays in the given program. That means printf() function is not used because in a Linux system, everything is a file, so the input of the keyboard is stored in a file. That is why printf() is not used. The program source code is shown below.

```c
/*
program description: calculates a factorial of a given number using
    iteration
*/

#define MAX 5

int main() /* main() doesn't have argument(s), which means a single path*/
{
  int i, num=i=MAX;
  /* we have a loop   */
  while (--i)
    num *= i;
  return num;
}
```

Listing 4.2: Calculating A Factorial of A Given Number

By analyzing the program-flow, there are two distinctive software characteristics, which has loop and single-path. The given program is categorized into a **SL** software characteristic set. The used equation of estimating execution time of a **SL** software characteristic set is shown below.

$$\begin{aligned} T(M, I, C, F, B, S_2, A, S_1) &= (2.47 \cdot 10^{-10}) \cdot M + (1.6 \cdot 10^{-10}) \cdot I \\ &+ (-4.4 \cdot 10^{-6}) \cdot C + (-1.8 \cdot 10^{-5}) \cdot F + (-1.3 \cdot 10^{-7}) \cdot B \\ &+ (-5.8 \cdot 10^{-7}) \cdot S_2 + (1.59 \cdot 10^{-8}) \cdot A + (-4.3 \cdot 10^{-8}) \cdot S_1 \end{aligned}$$

We already know what equation we need to use to estimate the execution time of the given program, and we know these parameters' values: **C**= 4, **F**= $2GHz$, **B**= 2 slots, $\mathbf{S_2}$= $4MB$, **A**= 4-way associativity, and $\mathbf{S_1}$= $8kB$.

The execution time, $\mathbf{T(M,I,C,F,B,S_2,A,\ S_1)}$, is missing two parameters, which are the memory access (**M**) and the number of instructions (**I**). These two parameters have a strong relation and depend on the complier optimization. There are several methods to measure the number of instructions (**I**), such as reading how many lines there are in the disassembler output. On the other hand, to calculate the values of memory access (**M**), we need to evaluate the memory instruction of the disassembler output.

We used GEM5 to have the number of instructions (**I**) and memory access (**M**) as follows:

M: 369920 bytes

I: 14107 instructions

Because we have all the needed parameters for execution time, $\mathbf{T(M,I,C,F,B,S_2,A,\ S_1)}$, the estimated execution time is shown as follows:

$$T(M, I, C, F, B, S_2, A, S_1) = T(369920, 14107, 4, 2, 2, 4, 4, 8) = (2.19 \cdot 10^{-10}) \cdot 369920$$
$$+(-4.08 \cdot 10^{-8}) \cdot 14107 + (0.2 \cdot 10^{-3}) \cdot 4 + (-1.6 \cdot 10^{-5}) \cdot 2 + (1.56 \cdot 10^{-7}) \cdot 2$$
$$+(-5.1 \cdot 10^{-7}) \cdot 4 + (1.67 \cdot 10^{-8}) \cdot 4 + (-5.6 \cdot 10^{-8}) \cdot 8 = 4.60144248E - \cdot 10^{-5}$$

According to the the execution time estimation algorithms for a **SL** software characteristic set, the estimated time for the given program running in the given hardware, is $3.7607525 \cdot 10^{-5}$ seconds. Table 4.17 shows the comparison between the estimated execution time and the conducted execution time.

Table 4.16: Comparison 1

Type	**Value**
Estimated Execution Time	$3.7607525 \cdot 10^{-5}$
Conducted Execution Time	0.000039

The error percentage of the estimated and conducted execution time is calculated as follows:

$$error = \frac{|T_{estimated} - T_{conducted}|}{T_{conducted}} \cdot 100\% = 3.57\%$$

The error is 3.57% for the estimation algorithm, which is very close to the actual conducted execution time result. The reason for this low error estimation percentage is because all the cores are running the same given program. In fact, running the same program in all the cores does not allow the memory access to change significantly.

We still want to use the given program, which calculates a factorial of a given number. The other cores are going to run these benchmarks, which are: matmult, prime, and statemate. Table 4.17 shows each core is mapped to a program.

Table 4.17: Mapping Each Executable To A Corresponding Core

Core Name	**Executable Name**
$\mathbf{C_0}$	The given executable (factorial)
$\mathbf{C_1}$	matmult
$\mathbf{C_2}$	prime
$\mathbf{C_3}$	statamate

We are just estimating the given program execution time, and not estimating the execution time for matmult, prime, and statamate. The hardware parameters and the software parameters are the same as before, which are: **C**$= 4$, **F**$= 2GHz$, **B**$=$ 2 slots, $\mathbf{S_2}$$= 4MB$, **A**$=$ 4-way associativity, $\mathbf{S_1}$$= 8kB$, **M**$=$ 369920 bytes, and **I**$=$ 17872. That would give the same execution time estimation as before, which is $\mathbf{T}(\mathbf{M,I,C,F,B,S_2,A,S_1}) = 3.7607525 \cdot 10^{-5}$. Table 4.18 shows the comparison between the estimated execution time and the conducted execution time for each core running a different program.

Table 4.18: Comparison 2

Type	**Value**
Estimated Execution Time	$3.7607525 \cdot 10^{-5}$
Conducted Execution Time	0.0000426

The error percentage of the estimated and conducted execution time is calculated as follows:

$$error = \frac{|T_{estimated} - T_{conducted}|}{T_{conducted}} \cdot 100\% = 11.72\%$$

The error is 11.72% for the estimation algorithm, which is very close to the actual conducted execution time result. The reason of this error is because the unpredictability of resource sharing, mainly the bus contention.

4.4 Summary

We implemented our automated test engine and employed it to $357,120$ experiments effectively. With these experimental results, we used robust regression algorithm to capture the relationship between execution times and computer architecture configurations. We then conducted extensive studies to further validate our approach. Through our experimental study, we found that the used validation method needs at least two benchmarks of the same software characteristics, but we also found that Malardalen has software characteristic sets that just have one benchmark in their set. The software characteristic sets that have more than a benchmark in their set, which are **L**, **LNA**, **LNAF**, **SL**, and **SLNAF** worked effectively in the validation tests. After we have examined their accuracy, we used them in an example to show the potential of the developed execution time estimation algorithms.

CHAPTER 5

Conclusions

The execution time is very important for computer system designs, especially for real-time systems. In fact, the accuracy of execution time estimation is significant for guaranteeing real-time system deadlines. For example, medical and avionic applications cannot fail to meet a specific deadline because they would have catastrophic consequences, such as loss of life or plane crash [21, 19]. Meeting all their deadlines is a must, failure to do so will result in severe consequences.

Significant research on execution time estimation has been conducted for programs running on single-core architectures. However, according to Wilhelm's Survey, the traditional execution time estimation suffers from computational cost, which comes from the difficulty of estimating the execution time for a given architecture, and model inflexibility, when applying the execution time estimation model in a different hardware architecture which commonly losses its effectiveness [40].

Our goal is deriving execution time estimation model that has fewer computations and better hardware architecture flexibility. We believe that there is a close relationship between the execution time and architecture features, and regression algorithm can accurately capture this relationship. Therefore, our approach is the following:

1. Develop a simulation platform and environment to facilitate the benchmark profiling and result analysis;

2. To improve the accuracy, we classify software characteristics into different subcategories;

3. Use robust regression algorithm to capture the execution time and hardware architecture configurations relationship;

4. Test effectiveness of the proposed approach.

We have evaluated the accuracy and precision for these **L**, **LNA**, **LNAF**, **SL**, and **SLNAF** execution time estimations that we developed from the automated test engine. The evaluation results are promising for estimating a program's execution time. The worst error that we have was less than 11.72%. Also, we found that the used validation method for execution time estimation does not support all the equations because the used validation method has a restriction on the sample number of benchmarks. As a result, we could not validate the memory access (**M**) and number of instructions (**I**) that have a significant variance with these sets: **LA**, **SLU**, **SLNAB**, **SLR**, **SLA**, **LB**, **SLNA**, **SLAB**, and **LAB**, because the adopted Malardalen's benchmarks just have one benchmark in these sets. We have to test all the execution time estimation dependent variables; therefore, we need at least two benchmarks that have similar software characteristics to do the used validation method.

The verified execution time estimations **L**, **LNA**, **LNAF**, **SL**, and **SLNAF** are shown again in this section.

$$T = \begin{cases} (8.4 \cdot 10^{-10}) \cdot M + (2.2 \cdot 10^{-10}) \cdot I + (-2.8 \cdot 10^{-5}) \cdot C + (-6.45 \cdot 10^{-5}) \cdot F + (-2.5 \cdot 10^{-7}) \cdot B + (-2.3 \cdot 10^{-6}) \cdot S_2 + (-9.7 \cdot 10^{-7}) \cdot A + (-2.9 \cdot 10^{-7}) \cdot S_1 & \textbf{L} \\ (2.8 \cdot 10^{-10}) \cdot M + (3.07 \cdot 10^{-10}) \cdot I + (-7.6 \cdot 10^{-6}) \cdot C + (-2.1 \cdot 10^{-5}) \cdot F + (1.6 \cdot 10^{-7}) \cdot B + (-6.6 \cdot 10^{-7}) \cdot S_2 + (1.1 \cdot 10^{-8}) \cdot A + (-4.9 \cdot 10^{-8}) \cdot S_1 & \textbf{LNA} \\ (2.5 \cdot 10^{-10}) \cdot M + (1.9 \cdot 10^{-10}) \cdot I + (-4.5 \cdot 10^{-6}) \cdot C + (-1.9 \cdot 10^{-5}) \cdot F + (1.45 \cdot 10^{-7}) \cdot B + (-5.9 \cdot 10^{-7}) \cdot S_2 + (1.43 \cdot 10^{-8}) \cdot A + (-7.7 \cdot 10^{-8}) \cdot S_1 & \textbf{LNAF} \\ (2.47 \cdot 10^{-10}) \cdot M + (1.6 \cdot 10^{-10}) \cdot I + (-4.4 \cdot 10^{-6}) \cdot C + (-1.8 \cdot 10^{-5}) \cdot F + (-1.3 \cdot 10^{-7}) \cdot B + (-5.8 \cdot 10^{-7}) \cdot S_2 + (1.59 \cdot 10^{-8}) \cdot A + (-4.3 \cdot 10^{-8}) \cdot S_1 & \textbf{SL} \\ (2.49 \cdot 10^{-10}) \cdot M + (3.58 \cdot 10^{-10}) \cdot I + (-5.5 \cdot 10^{-6}) \cdot C + (-1.89 \cdot 10^{-5}) \cdot F + (1.68 \cdot 10^{-7}) \cdot B + (-5.8 \cdot 10^{-7}) \cdot S_2 + (1.85 \cdot 10^{-8}) \cdot A + (-7.8 \cdot 10^{-8}) \cdot S_1 & \textbf{SLNAF} \end{cases}$$

REFERENCES

[1] gem5. page www.gem5.org.

[2] Mlardalen wcet benchmark. page http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[3] Simplescalar llc. page http://www.simplescalar.com/.

[4] Statsmodels. page http://statsmodels.sourceforge.net/.

[5] M. Abd-El-Barr and H. El-Rewini. Fundamentals of computer organization and architecture. pages 107 – 187, 2010.

[6] T. Austin, E. Larson, and D. Ernst. Simple scalar: an infrastructure for computer system modeling. *IEEE computer society*, 35, 2002.

[7] G. Bernat, A. Colin, and S. Petters. Wcet analysis of probabilistic hard real-time systems. *RTSS*, 2002.

[8] G. Binkert, N. Reinhardt, K, and Saidi. Processor and system-on-chip simulation. *IEEE*, 5, 2010.

[9] N. Binkert and B. Beckmann. The gem5 simulator. *ACM SIGARCH Computer Architecture*, 39, 2011.

[10] A. Bivens. Architectural design for next generation heterogeneous memory systems. *IEEE International Memory Workshop*, pages 1 – 4, 2010.

[11] A. Bivens. Architectural design for next generation heterogeneous memory systems. *IEEE International Memory Workshop*, 53:1 4, 2010.

[12] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimization. *ISCA*, 6, 2000.

[13] P. Burgio, M. Ruggiero, and F. Esposito. Adaptive tdma bus allocation and elastic scheduling: A unified approach for enhancing robustness in multi-core rt systems. *IEEE*, page 187 191, 2010.

[14] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli. Accuracy evaluation of gem5 simulator system. *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, pages 1 – 7, 2012.

[15] X. Caramanis and S. Mannor. Robust regression and lasso. *IEEE Transactions on Information*, page 35613574, 2010.

[16] T. Chantem, S. Hu, and R. Dick. Temperature-aware scheduling and assignment for hard real-time applications on mpsocs. *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS*, 19(10):2 5, 2011.

[17] C. Chen. Robust regression and outlier detection with the robustreg procedure. *SUGI 27*, pages 265 – 270, 2010.

[18] P. Crowley and J. Baer. Worst-case execution time estimation for hardware-assisted multithreaded processors. *IEEE*, 2005.

[19] R. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR)*, 43(35):32 37, 2011.

[20] D. Ghosh, J. Sheehy, K. K. Thorup, and S. Vinoski. Programming language impact on the development of distributed systems. *Journal of Internet Services and Applications*, 3(1):25 – 29, 2012.

[21] Y. Gu and D. Jin. Drop test simulation and doe analysis for design optimization of micro-electronics packages. *IEEE Electronic Components and Technology Conference*, page 6, 2006.

[22] G. Hellestrand. The engineering of supersystems. *IEEE Computer*, 38:103 – 105, 2005.

[23] P. Hollanda and R. Welschb. Robust regression using iteratively reweighted least-squares. 6(9):813 – 827, 2007.

[24] M. Hsieh, K. Pedretti, and J. Meng. Sst + gem5 = a scalable simulation infrastructure for high performance computing. *SIMUTOOLS '12 Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, pages 196–201, 2012.

[25] C. L. Janssen, H. Adalsteinsson, and S. Cranford. A simulator for large-scale parallel computer architectures. *ACM SIGARCH Computer Architecture News archive*, 1(2):15 – 16, 2010.

[26] K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. *IEEE Real-Time Systems Symposium*, 31:3 8, 2010.

[27] M. E. Latoschik and H. Tramberend. Simulator x: A scalable and concurrent architecture for intelligent realtime interactive systems. *IEEE Virtual Reality Conference (VR)*, pages 171 – 174, 2011.

[28] Y. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *DAC*, pages 456 – 461, 1995.

[29] H. Lv, Y. Cheng, Y. Xiaoxi, G. Xiaotong, and Z. Weihua. P-gas: Parallelizing a cycle-accurate event-driven many-core processor simulator using parallel discrete event simulation. *PADS '10 Proceedings of the 2010 IEEE Workshop on Principles of Advanced and Distributed Simulation*, pages 94 – 96, 2010.

[30] C. McGrath, K. Ahmed, and P. Conway. The amd opteron processor for multiprocessor servers. *IEEE Micro 23*, 2:66 76, 2003.

[31] J. Mogul, A. Baumann, and T. Roscoe. Mind the gap: reconnecting architecture and os research. *HotOS'13 Proceedings of the 13th USENIX conference on Hot topics in operating systems*, 2011.

[32] R. Parasuraman and D. Manzey. Complacency and bias in human use of automation: An attentional integration. *IEEE*, page 7 8, 2010.

[33] A. Patel, F. Afram, S. Chen, and K. Ghose. Marssx86: A full system simulator for x86 cpus. *DAC'11*, 2, 2011.

[34] R. Pellizzoni and L. Sha. Coscheduling of cpu and i/o transactions in cots-based embedded systems. *Proceedings of the 2008 Real-Time Systems Symposium, RTSS '08*, 2008.

[35] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE COMPUTER ARCHITECTURE LETTERS*, 10(1):18 21, 2011.

[36] H. Shah and A. Raabe. Challenges of wcet analysis in cots multi-core due to diferent levels of abstraction. *IEEE*, 2013.

[37] D. Staiculescu, N. Bushyager, A. Obatoyinbo, L. Martin, and M. Tentzeris. Design and optimization of 3-d compact stripline and microstrip bluetooth/wlan balun architectures using the design of experiments technique. *IEEE Trans. Antennas Propag*, 53(5):18051812, 2005.

[38] S. Thesing, J. Souyris, and R. Heckmann. An abstract interpretation-base timing validation avionics software systems. *DNS*, pages 625 – 632, 2003.

[39] M. Thuresson, M. Sjlander, and M. Bjrk. Flexcore: Utilizing exposed datapath control for efficient computing. *IEEE: Journal of Signal Processing Systems*, 57(1):7 11, 2009.

[40] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, and J. Staschulat. The worst-case execution time problem overview of methods and survey of tools. 2008.

[41] W. Wolf. High performance embedded computing. *San Francisco: Morgan Kaufmann*, (3), 2001.

[42] Y. Xiaoxi, G. Xiaotong, and Z. Weihua. Analysis and comparison of a few architecture simulator acceleration technologies. *IEEE*, 2011.

[43] K. Yang, Y. Fu, X. Han, and J. Jiang. Efficient broadcast scheme based on subnetwork partition for many-core cmps on gem5 simulator. *Computer Engineering and Technology Communications in Computer and Information Science*, 337:163 – 172, 2012.

[44] W. Yang and H. Xu. A unified robust regression model for lasso-like algorithms. *IEEE Transactions on Information*, page 35613574, 2013.

[45] H. Zeng, M. Yourst, and K. Ghose. Mptlsim: a cycle-accurate, full-system simulator for x86-64 multicore architectures with coherent caches. *ACM SIGARCH Computer Architecture News archive*, 37(2):4 – 9, 2009.

[46] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. *In Proceedings of the ISCA 2009*, pages 14 – 23, 2009.