HYBRID PROBABILISTIC TIMING ANALYSIS WITH EXTREME VALUE THEORY AND COPULAS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

LEVENT BEKDEMİR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

SEPTEMBER 2019

Approval of the thesis:

HYBRID PROBABILISTIC TIMING ANALYSIS WITH EXTREME VALUE THEORY AND COPULAS

submitted by **LEVENT BEKDEMİR** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** ______________

Prof. Dr. İlkay Ulusoy
Head of Department, **Electrical and Electronics Engineering** ______________

Assoc. Prof. Dr. Cüneyt Fehmi Bazlamaçcı
Supervisor, **Electrical and Electronics Engineering, METU** ______________

Examining Committee Members:

Prof. Dr. İlkay Ulusoy
Electrical and Electronics Engineering Dept., METU ______________

Assoc. Prof. Dr. Cüneyt Fehmi Bazlamaçcı
Electrical and Electronics Engineering Dept., METU ______________

Prof. Dr. Gözde B. Akar
Electrical and Electronics Engineering Dept., METU ______________

Prof. Dr. Ece G. Schmidt
Electrical and Electronics Engineering Dept., METU ______________

Assoc. Prof. Dr. Süleyman Tosun
Computer Engineering Dept., Hacettepe University ______________

Date:

ABSTRACT

HYBRID PROBABILISTIC TIMING ANALYSIS WITH EXTREME VALUE THEORY AND COPULAS

BEKDEMİR, Levent
M.S., Department of Electrical and Electronics Engineering
Supervisor: Assoc. Prof. Dr. Cüneyt Fehmi Bazlamaçcı

September 2019, 167 pages

The primary challenge of time-critical systems is to ensure that a task completes its execution before its deadline. In order to ensure that the underlying system comply with stringent timing requirements, designers ought to analyze the timing behavior of the software and its sub-components. Worst-Case Execution Time (WCET) represents the maximum length of time an individual software unit takes to execute and is the most essential value for schedulability analysis in safety-critical systems. Recent studies focus on statistical approaches which augments measurement-based timing analysis with probabilistic confidence level by applying stochastic methods.

Common approaches either utilize Extreme Value Theory(EVT) for end-to-end measurements or convolution techniques for a group of program units to derive absolute upper distributional bound of the whole program. The former method lacks insurance of path coverage while the latter one suffers from ignoring possible extreme cases of program units. Furthermore, current state-of-the-art convolution method that is being implemented by a commercial WCET analysis tool overestimates the results under the assumption of worst dependence between the basic blocks.

In this thesis, we propose a hybrid probabilistic timing analysis framework based on modeling the program units with EVT to capture extreme cases and Copulas to model the dependency between the units to derive tighter distributional bounds to mitigate the effects of comonotonic assumptions. The proposed framework also offers a way to minimize the instrumentation probe effects which is essential to obtain fine-grained execution time traces on COTS platforms.

Keywords: Worst-Case Execution Timing Analysis, Measurement-Based Probabilistic Timing Analysis, Copula Theory, Extreme Value Theory, Minimum Probe Effect

ÖZ

UÇ DEĞER TEOREMİ VE KOPULA İLE HİBRİD OLASILIKSAL ZAMANLAMA ANALİZİ

BEKDEMİR, Levent
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Cüneyt Fehmi Bazlamaççı

Eylül 2019 , 167 sayfa

Zaman kritik sistemlerde birincil zorluk, bir görevin işlevini tanımlı zaman sınırları dahilinde tamamlayabilmesidir. Sistem için tanımlanmış olan katı zamanlama gereksinimlerini karşılayabilmek için tasarımcılar tarafından yazılım ve yazılım alt bileşenlerinin zamanlama analizlerinin gerçekleştirilmesi gerekmektedir. Yazılım birimlerinin fonksiyonlarını gerçekleştirirken harcadıkları en uzun zamana en kötü durum yürütme süresi denmektedir ve bu değer emniyet kritik sistemlerin zamanlama analizi için en önemli girdidir. Son yıllarda yapılan çalışmalar istatistiksel metotların ölçüm tabanlı en kötü durum yürütme süresi analizi çalışmaları üzerinde uygulanmasına odaklanmaktadır.

Bu çalışmalarda uygulanan genel yaklaşım programların uçtan uca alınan ölçüm değerleri üzerinde uç değer teoreminin uygulanması veya küçük program bileşenlerinden toplanan ölçüm değerleri üzerinde konvolusyon işlemi uygulanarak tüm programın en kötü durum yürütme süresi dağılımının üst limitini tahmin etmek olarak ikiye ayrılmaktadır. Bahsedilen ilk yöntem program yol kapsamasını garanti etmemekte, ikinci yöntem ise olası uç değerleri göz ardı etmektedir. Ayrıca, mevcut durumda ticari bir analiz aracında uygulanmakta olan gelişmiş konvolusyon yöntemi, program birimleri arasındaki bağımlılığın en kötüsü olduğunu varsayarak sonuçların olması gerekenden fazla hesaplanmasına sebep olmaktadır.

Bu tezde, uç değerlerin göz ardı edilmemesi için program birimlerinin yürütme sürelerinin uç değer teoremi ile modellenmesi ve kopulalar yardımıyla program birimleri arası bağımlılığın modellenmesini sağlayan hibrid olasılıksal zamanlama analizi çatısı önerilmektedir. Böylece gelişmiş konvolusyon yönteminde fazla hesaplamaya sebep olan etkiler azaltılmakta ve daha daraltılmış bir üst limit dağılımı elde edilmektedir. Önerilen bu çatı ayrıca rafta hazır ürünler üzerinde ölçüm tabanlı analizlerde yapılan kod değişiminin sebep olduğu prob etkilerini en aza indirecek bir yöntem sunmaktadır.

Anahtar Kelimeler: En Kötü Durum Yürütme Süresi, Ölçüm Tabanlı Olasılıksal Zamanlama Analizi, Kopula Teorisi, Uç Değer Teoremi, Minimum Prob Etkisi

To my family...

ACKNOWLEDGMENTS

This study is the result of a huge sacrifice and commitment in order to prove myself that a determined mind can overcome any challenges. During this long journey, I would first like to thank my thesis advisor Assoc.Prof.Dr. Cüneyt Fehmi Bazlamaçcı, for his continuous support and guidance and most importantly his endless mental support. Your encouragement throughout this period helped me to cross the finish line.

I would like to thank my colleague and chief engineer Akın Yılmaz in Turkish Aerospace Industries Inc. for his technical advices and his tolerance to my absence during this period. I also thank to my friends Veli and Bora for sharing their experiences with me. Most importantly thank you for accompanying me during my grumpy moments.

A special thanks to my family especially my mother for their permanent support in this challenge. They were there for me whenever I needed them and it was always relieving to talk to them on the phone. My mother never gave up on me and always encouraged me to finish what I started. Thank you all for your faith and love.

Finally, Thanks to Melike, my beloved wife, for being an amazing life partner. Thank you for encouraging me to pursue my goals in any situation, supporting me in any condition and for being the responsible for my very joyful moments. Thank you for all your love and patience no matter what. Thank you for sharing and building dreams with me. Thank you for literally sitting next to me while I was studying to preserve my focus on this thesis. I love you with all my heart.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ABBREVIATIONS	
AD	Anderson-Darling
AHB	The Advanced High-Performance Bus
ATD	Analysis-Time Distribution
ATS	Analysis-Time Samples
BCET	Best-Case Execution Time
BM	Block Maxima
CAN	Controller Area Network
CCDF	Complementary Cumulative Distribution Function
CDF	Cumulative Distribution Function
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CRPS	Continuous Probability Rank Score
DSU	Debug Support Unit
ECCDF	Empirical Complementary Cumulative Distribution Function
ECDF	Empirical Cumulative Distribution Function
EPC	Extended Path Coverage
ET	Exponential Test
ETP	Execution Time Profile
EV	Extreme-Value
EVT	Extreme-Value Theory
FPGA	Field-Programmable Gate Array
FPU	Floating-Point Unit
GAEP	Generalized AEP Algorithm

GDB	GNU Debugger
GEV	Generalized Extreme-Value
GMLE	Generalized Maximum Likelihood Estimation
GPD	Generalized Pareto Distribution
GPIO	General Purpose Input/Output
HOET	Highest Observed Execution Time
HPS	Hard Processor System
HYPTA	Hybrid Probabilistic Timing Analysis
i.i.d	Independent and Identically Distributed
I/D	Instruction/Data
I/O	Input/Output
ICDF	Inverse Cumulative Distribution Function
IMA	Integrated Modular Avionics
KS	Kolmogorov-Smirnov
LB	Ljung-Box
LRU	Least Recently Used
MBPTA	Measurement-Based Probabilistic Timing Analysis
MBTA	Measurement-Based Timing Analysis
MLE	Maximum Likelihood Estimation
MMU	Memory Management Unit
MSc	Master of Science
MSGDMA	Modular Scatter-Gather Direct Memory Access
OTD	Operation-Time Distribution
PCI	Peripheral Component Interconnect
PDF	Probability Distribution Function
PhD	Doctor of Philosophy
PLL	Phase-Locked Loop

PoT	Peaks Over Threshold
PTA	Probabilistic Timing Analysis
PUB	Path Upper Bounding
pWCET	Probabilistic Worst-Case Execution Time
QQ	Quantile-Quantile
RAM	Random Access Memory
RTEMS	Real-Time Executive for Multiprocessor Systems
RTOS	Real-Time Operating System
SDRAM	Synchronous Dynamic Random Access Memory
SETV	Sources of Execution Time Variability
SoC	System-on-a-chip
SoJ	Source of Jitter
SPD	Semi-Parametric Piecewise Distribution
SPTA	Static Probabilistic Timing Analysis
SSH	Secure Shell
TAI	Turkish Aerospace Industries, Inc.
UART	Universal Asynchronous Receiver-Transmitter
VaR	Value at Risk
VHDL	VHSIC Hardware Description Language
WCET	Worst-Case Execution Time
WW	Wald-Wolfowitz

CHAPTER 1

INTRODUCTION

1.1 Motivation and Problem Definition

The most distinguishing characteristic of a real-time system is to give correct response within its strictly defined deadline. Systems range from personal computing devices, cellphones, robots, industrial systems to safety-critical systems such as autonomous vehicles, flight control systems, and satellite control systems contain examples of real-time applications. In each domain, safety-critical real-time applications are subject to a certain degree of safety-related software standards such as DO178B/C(Avionics) [5], ISO26262(Automotive) [6], EN50128(Railway) [7], and ECSS-E-ST-40C(Space) [8]. In order to satisfy safety related requirements of these standards, timing characteristics of software units must be estimated to quantify safety confidence along with the system level requirements.

Decades of research efforts have been devoted to improve computing power. Architectural optimizations and software abstractions increased the overall performance and reduced the necessary development and maintenance efforts. However, advances in technology arose challenges in timing analysis. Performance enhancing features of modern complex processors such as multi-cores, shared buses, pipelines, out-of-order execution, branch prediction, and caches made the execution time dependent to the execution history. These architectural improvements made static analysis techniques which rely on abstract hardware models inadequate because of the lack of knowledge of the complete architecture, hence requiring more user interaction.

To bound the execution time of a task, common approach in industry is to obtain several measurements and add a safety margin usually about %20 which is scientifically inaccurate. In order to obtain sound estimates with measurement-based techniques, the software needs to be executed under all possible conditions with all possible operation states which is intractable in practice. Furthermore, measurement-based methods require an instrumentation code that is to be injected into the program yielding probe effects.

Increased complexity in hardware and software causes jittery response times. The variability in execution times makes statistical methods to be applicable in worst-case execution time analysis. Statistical analysis provides WCET estimates with increased confidence without the necessity of full path coverage, thus allowing us to obtain only a few measurements to estimate the worst timing behavior of the system.

Some works have focused on applying algebra of probability distributions (convolutions) to estimate the WCET. Those hybrid approaches aim to combine the structural information of the analyzed software with the probability distributions of the measurements of small program units. Convolution of those small blocks yields an execution time distribution which represents the execution time of different program paths. Thus, convolving the blocks virtually generates possible paths that are not covered during the test runs. However, some of those convolution approaches generate new paths that might be infeasible to reach in reality which leads to overestimation in the results.

Extreme Value Theory(EVT) is another emerging method in probabilistic timing analysis domain which can derive estimates for extreme cases. EVT aims to model the extreme tails of a distribution by fitting a probability distribution to estimate a WCET with arbitrarily low exceedance probability. EVT is considered reliable only if the samples given are independent and identically distributed (i.i.d). This assumption might not hold for simple architectures with simple software, but advanced architectures which generates noise on execution times and complex software with interrupts, queues, semaphores, etc. makes it possible to analyze through EVT. However, most of the studies in literature applies EVT to end-to-end measurements of the analyzed programs which suffers from the path coverage problem during the analysis runs.

Usage of Copulas in Timing Analysis is first and only studied by G. Bernat, A. Burns, and M. Newby in [9]. In this study they proposed a framework by using copulas for probabilistic hard real-time systems. Throughout the case study in that paper, they mostly examined the Fréchet upper and lower bounds of the copulas and concluded that the assumption of comonotonicity is the only safe but pessimistic choice. However, the application procedure to derive the empirical Copulas, determination of the distributions of the marginals, a computationally tractable solution to estimate the joint distribution through empirical Copulas and extrapolation of the unobserved tail behaviors for the marginals were missing in that paper since the paper only focuses on the theory of using Copulas in the WCET domain. In fact, they addressed these missing points as the future works. The idea of integrating EVT and Copulas within a unified framework in the present thesis study stemmed from those future work ideas. Hence, that paper represents the present basis of this thesis study.

Our team at Turkish Aerospace Industries, Inc. (TAI), has been developing a Satellite Flight Software with Integrated Modular Avionics (IMA) concept. Low-level abstraction layers have been developed such as Operating System Abstraction Layer and Hardware Abstraction Layer so that developers can focus only on the Application Layer. Underlying hardware is a SPARC V8 based LEON3 processor with 7-stage pipeline and I/D Caches. On top of the Application Layer, Component-Based software development approach has been adapted, thus scheduled software units are named as Components in our system. In order to ensure that the time-critical components comply with their temporal requirements an industrially-viable and trustworthy framework is needed.

The only option on the market today based on measurement-based timing analysis is the RapiTime tool by Rapita Inc. The tool makes use of the hybrid probabilistic timing analysis approach with conservative convolution method. During the trials it seemed to overestimate the results by a huge factor. Consequently, an improvement on top of their approach is needed to decrease the overestimation, but staying in the safe side not to underestimate the results by relying on the statistical evidences. In addition to that, a mechanism to get EVT involved for capturing extreme events of program units would increase the trustworthiness of the results.

1.2 Proposed Methods and Models

The aim of this study is to provide a hybrid probabilistic measurement based timing analysis framework for COTS platforms that are being used in the industry. Our proposed method specifically targets COTS platforms rather than customized ones.

In order to comply with the EVT requirements and decrease the probe effect resulting from the instrumentation, software components are divided into functional blocks. It is seen from the demonstrations that dividing the program into basic block granularity leads to unacceptable probe effects in COTS platforms. Thus, analyzing the programs with functional block granularity generates less trace data which eases the analysis phase. Furthermore, functional blocks tend to be more input dependent and multi-path allowing EVT to be applicable.

In our proposed framework, injected instrumentation code is a collection of General Purpose I/O (GPIO) driving instructions which is the most non-intrusive mechanism across the usage of other possible external interfaces on-board. Capturing measurements are done through a custom developed external hardware which monitors the GPIO pins and time stamps the changes.

Dividing the program into functional blocks also allows to extract static structural information of the analyzed program. This structure information with the collected measurements are used to construct a probabilistic WCET (pWCET) distribution by virtually generating new paths to upper bound all possible execution scenarios. Calculation of the pWCET distribution is basically sum of n random variables from a mathematical point of view. Random variables represent the execution time of each functional block and sum of them represents the overall execution time of the program. Current state-of-the-art solutions either assume an independence between the blocks or use a conservative convolution approach named as *biased convolution* to upper bound all possible dependence types between the blocks. However, neither standard convolution(independence) nor the biased convolution reflects the real behavior of the programs. In fact, it is possible to model the dependence between the random variables by using Copulas. Copulas are basically joint probability distribution functions that have uniform marginals. Hence, it allows to simulate the joint behavior of the random variables which eases to derive the probability distribution of their sum. Consequently the overestimation or underestimation resulting from the assumptions of independence or comonotonic dependence are mitigated.

Additionally, current state-of-the-art method constructs the random variables from the observed measurements. As our proposed method divides the programs into functional blocks which tend to be multi-path sub-programs, it might be possible to model those functions with Extreme Value Theory, thus allowing us to predict possible rare events based on its tails behavior.

In the hybrid framework proposed in this work, software components are executed several times with different inputs while ensuring functional block coverage and measurement sets are collected for each function. By taking extracted structure of the program into consideration, copulas are fit between the execution time random variables of functional blocks. A monte-carlo simulation approach is used to simulate the possible upcoming behavior of the functional blocks which together represent the possible upcoming behavior of the program. While doing that, each marginal of the n-dimensional probability distribution are represented by a parametric continuous Extreme Value distribution whenever possible.

1.3 Contributions and Novelties

The contributions of this thesis are as follows:

- The identification of a hybrid timing analysis framework for industrial COTS platforms.
- Application procedures, limitations and assumptions of Measurement-Based Probabilistic Timing Analysis on an industrial COTS platform are given.
- An improvement over the current state-of-the-art commercial solution which is based on a conservative convolution mechanism [10] is detailed as the main scope of this study. Current solution assumes a worst type of dependence between the random variables that represents the execution time of basic blocks.

We model the dependence between the random variables which represent the execution time of functional blocks by using Copulas.

- A Monte-Carlo simulation approach that is widely used in economics [11] is adapted to simulate the execution of the individual functions inside the program which together simulates the program itself.

- Current commercial hybrid solution ignores the possible rare cases of individual software blocks. We introduce a mechanism to represent the execution time of functional blocks inside the program with Extreme Value distributions to capture the extreme events.

- Results of the case studies are compared with the results of the RapiTime tool and widely accepted independent assumptions. It can be seen from the results that our approach decreases the overestimation by providing tighter bounds and benefit from EVT whenever possible.

- We also introduce a way to reduce the instrumentation probe effect by minimizing the injected code size and using an external custom developed FPGA hardware named as TraceBox. Its design details and improvements over widely used online trace storage method are shown.

- All the timing analysis implementations are done in MATLAB and R environments. TraceBox hardware is implemented on an FPGA environment with VHDL language. All the source codes and design details are provided.

1.4 The Outline of the Thesis

The remainder of this study is organized as follows: chapter 2 presents the background information, theoretical notions of WCET analysis and literature review. Application procedure of EVT in COTS platforms and the main analysis test bench for the proposed hybrid framework are introduced in chapter 3. Chapter 4 presents the details of custom developed TraceBox hardware which is the essential part of this study to capture measurements with minimal probe effect. Proposed hybrid framework with EVT and Copulas are introduced and the results of the case studies are given in the Chapter 5. Chapter 6 gives the summary of this thesis and the possible extensions are discussed in Chapter 7.

CHAPTER 2

BACKGROUND AND LITERATURE OVERVIEW

2.1 BACKGROUND INFORMATION

2.1.1 Timing Analysis

A system is defined as real-time system as long as it satisfied the non-functional timing constraints. However, proving that requirement is not a trivial task. The easiest and the safest way to find the longest execution time of a program is to measure the execution time by giving the worst-case input to the program. In reality, knowing and providing worst-case conditions for the program is not possible [1]. Because of this obscurity, there have been numerous studies about Worst-Case Execution Time (WCET) analysis.

Figure 2.1 illustrates fundamental terms in WCET analysis domain. A program, task or the system as a whole shows different execution times according to its given input. In the figure, lower white filled curve represents measured execution times of a program under several conditions, it is also the subset off all possible execution times of the program which is illustrated by upper black filled curve. If it was possible to plot that upper curve, its starting and ending points in horizontal axis would give the Best-Case Execution Time (BCET), and WCET respectively. Studies in this domain focuses on finding a safe and tight upper-bound for the actual WCET which is an *estimation* in general case.

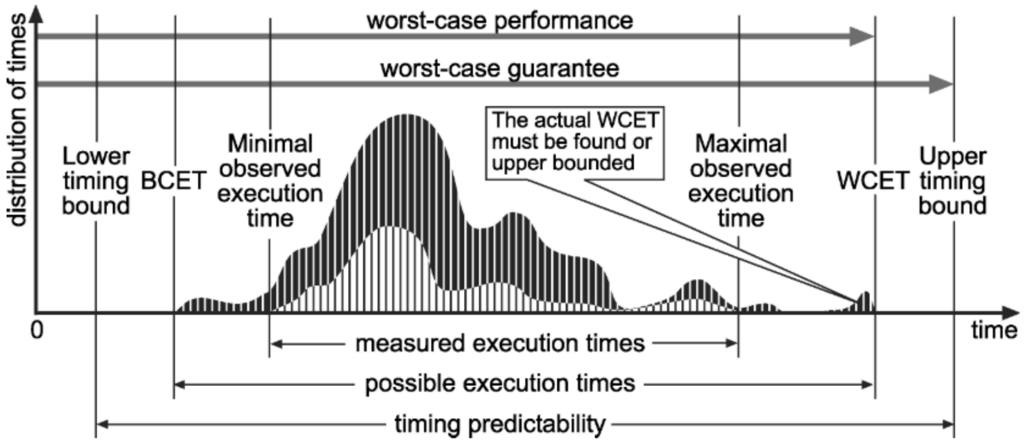


Figure 2.1: Timing Analysis Notions [1]

Although there are vast amount of methods in literature, each belongs to either static or dynamic timing analysis category. Static methods basically try to analytically calculate a safe upper bound for the actual WCET. These techniques do not rely on executing the program on a real hardware target, on the contrary they analyze the program code or executable object statically [12]. However, this method relies heavily on the presence of underlying hardware model. Architectural improvements in processors recently challenge these approaches which cause them to pessimistically overestimate WCET upper-bounds or make the cost of computation prohibitive.

The most preferred solution in industry is the end-to-end measurement method which is also called as dynamic timing analysis [1]. In this approach, programs are tested under some pre-determined conditions and measurements are taken from the system. Observed execution times construct the lower white filled curve in Figure 2.1 in which minimal and maximal observed execution times are detected. Maximal observed execution time is also referred as Highest Observed Execution Time (HOET) and the WCET is calculated by adding a 20% margin to the HOET [13]. This approach is not based on any scientific justification, but it works in many cases as long as sufficiently enough conditions are tested. Wilhelm et al. [1] states that measurement-based analysis results provide actual variability of the execution time and it can also be used as a validation for static analysis methods. Those results should not be far lower then the static analysis results which would indicate that the latter are imprecise.

Both static and dynamic timing analysis methods have some limitations and drawbacks. Static methods generally suffer from limited access to intellectual property and complexity of current processor architectures, thus, making static analysis techniques inaccurate or overly pessimistic. Since the main scope of this thesis is measurement-based methods, we are not going to give further details about static methods.

On the other hand, the most challenging issue of measurement-based (dynamic) timing analysis methods is the lack of knowledge about the coverage during analysis runs. A cartesian product of possible program inputs and initial hardware states are not feasible to cover in tests. Besides, controlling hardware states in isolation is not possible for commercial products. Although, it is noted that adding 20% over the HOET works for the most cases, a scientific justification is needed.

Recently, probabilistic timing analysis (PTA) methods have emerged in order to mitigate some of the drawbacks of the existing solutions. Static Probabilistic Timing Analysis (SPTA) techniques seek to reduce the pessimism due to the incomplete information about underlying platform by expressing some of its behaviors probabilistically. Measurement-Based Probabilistic Timing Analysis (MBPTA) on the other hand, seeks to scientifically reason about the worst case events that are are captured during analysis phase by modeling the execution time behavior of the program based on widely accepted statistical methods. Output of the MBPTA is not a single valued WCET, but rather a probability distribution of execution time profile of the program that is guaranteed to upper bound analysis time observations. Since, this approach is based on observations as in conventional MBTA methods, it still relies on the quality of the test scenarios.

2.1.2 Extreme Value Theory

Before going further, we first present the concept of Extreme Value Theory (EVT) since EVT is the building block of MBPTA [14].

Extreme Value Theory (EVT) is a branch of statistics that was designed to predict unusual natural events such as extreme floods, tornado outbreaks, and earthquakes. These extreme events is modeled as probability distributions and EVT deals with the extreme deviations from the median of observations associated to phenomena of interest.

EVT can be though analogues to Central Limit Theory: while the latter focuses on computing the mean of the normally distributed populations, EVT only concerns the tail behavior of a given distribution.

In order to obtain trustworthy results from EVT, it requires analyzed observations are represented as identically distributed independent (i.i.d) random variables. This property can be verified by some statistical tests on analyzed data which will be explained later. Normally, EVT itself cannot detect whether the analyzed data represents the possible upcoming rare events so the results obtained from EVT is only valid for samples obtained from the domain of attraction. Assuming the system as a black box, EVT studies the tail of the distribution of observations taken from the system to predict the occurrence of rare events with confidence.

According to [15] three major steps are taken when using statistics in any field which are (1) obtaining data, (2) mathematically formulating or fitting a proper model for the data, and (3) generalize the behavior of the system by using the fitted model. In line with [16] and [17], a typical implementation of EVT is described step-by-step in upcoming subsections.

2.1.2.1 Applicability Evidence

Firstly, it is necessary to obtain samples from the system which are large enough to be representative of the analyzed event but is still as small as possible to minimize the analysis cost. Furthermore, i.i.d random variable requirement of EVT is checked in this step to determine whether the obtained samples will be accepted, which is called as Applicability Evidence in literature.

In order to show that the i.i.d requirements are met, statistical hypothesis tests are employed. These tests are based on a null hypothesis H_0 which is accepted unless an adequate evidence is found to reject it, and an opposite hypothesis H_1 which is accepted if only if H_0 is rejected. Statistical tests produce p-values which represents the strength of the evidence against the null hypothesis which is a number in [0,1) range.

Statistical tests may produce false positives or false negatives which are controlled by determination of a significance level α which limits the probability of false negatives. α is typically kept between [0.01, 0.05], thus the decision result from statistical test is trusted with a confidence level γ = 1 – α. A p-value smaller than α indicates there is a strong evidence against the null hypothesis so H_0 is rejected. Likewise, a p-value higher than α indicates a weak evidence against H_0, so you fail to reject H_0. There are several statistical tests to prove that the random variables are identically distributed and independent such as Wald-Wolfowitz (WW) and Ljung-Box (LB) for independency, and Kolmogorov-Smirnov (KS) and k-sample Anderson-Darling (AD) tests for identicality. The application of these tests produce p-values, which are expected to be greater than determined α and are uniformly distributed in [0, 1) interval, hence proving that the observations (1) were not produced by a non-random process, (2) do not present a relevant dependency, or (3) were not drawn from different distributions.

2.1.2.2 Data Selection

An extreme value refers to very small or very large values laying on the tails of the probability distributions. These extremely high or small values in a data set constructs the behavior of the tails of the distribution so in order to approximate a possible lower or higher value from the observations, irrelevant data must be eliminated.

Relevant maximum values can be selected either by using Block Maxima (BM) or Peaks Over Threshold (PoT) approaches in the domain of EVT [18]. BM approach divides the data into equal size blocks and gets the maximum of each block, thus the resulting data set represents the higher tail of the distribution. In PoT approach, only the values exceeding a certain threshold are selected to represent the tail. Figure 2.2 illustrates BM and PoT approaches. Selection of either method depends on the application area. While using both approaches, block size or exceedance threshold should be carefully selected which will be explained in the following sections.

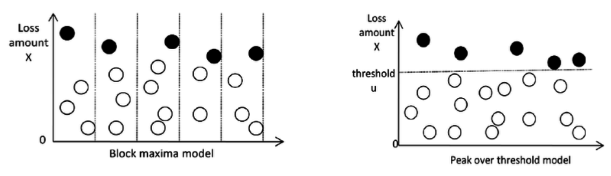


Figure 2.2: Block Maxima and Peaks over Threshold Approaches

2.1.2.3 Model Fitting

The model that represents the observations in EVT domain is called as Extreme Value Distribution in general. The distribution simply models how large (or small) the observed data will probably get.

Without using statistical models, it is possible to obtain some samples and construct an empirical distribution function $F(x)$ and by using its inverse one can calculate $w = F^{-1}(1 - p_e)$ where p_e corresponds to the probability of exceedance and w is the estimated value with p_e. However, it is not possible to estimate higher values than already observed highest value with this approach [19]. Furthermore, a vast amount of observations must be made i.e to estimate w with $p_e = 10^{-n}$, 10^n samples are needed.

When $Y = max(X_1, X_2, ..., X_n)$ is the random variable which is constructed by BM, EVT tells that distribution of Y will converge to Generalized Extreme Value Distribution (GEV) [20].

Generalized Extreme Value distribution is a three-parameter (ξ, μ, σ) distribution which is capable of representing Gumbel, Fréchet and Weibull distributions depending on the shape (ξ) parameter and The Cumulative Distribution Function (CDF) of GEV is defined as follows [20]:

$$G(x; \mu, \sigma, \xi) = exp\left[-\left(1 + \xi\frac{x-\mu}{\sigma}\right)^{-1/\xi}\right] \tag{21}$$

Gumbel ($\xi = 0$): The distribution has exponential tail and it is unbounded.

Fréchet ($\xi \geq 0$): It has heavy tail and decreases polynomially.

Weibull ($\xi \leq 0$): The distribution has light tail with a sharp slope and has a maximum value.

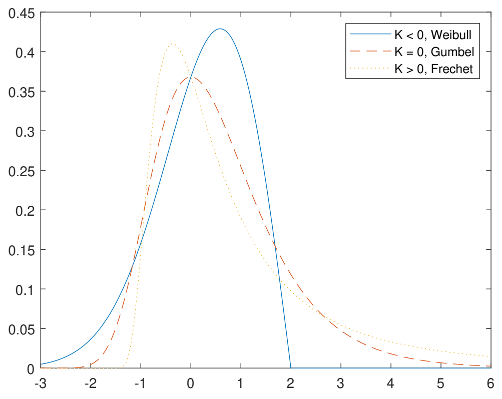
When $Y = \{X_i - u | X_i > u\}$ where u is the threshold which means Y is constructed by PoT, the distribution converges to Generalized Pareto Distribution (GPD) [20]. The CDF of GPD is defined as:

When $\mu = 0$,

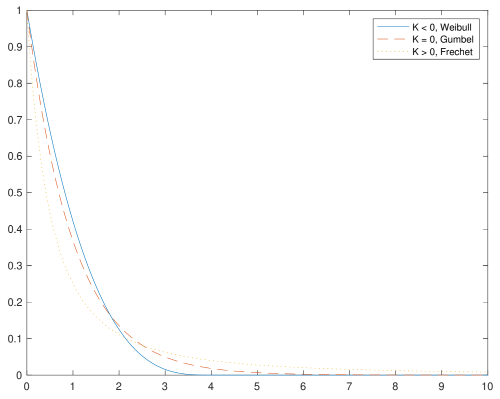
$$G(x; \sigma, \xi) = \begin{cases} 1 - \left(1 + \xi \frac{x}{\sigma}\right)^{-1/\xi} & \text{if } \xi \neq 0 \\ 1 - e^{-x/\sigma} & \text{if } \xi = 0 \end{cases} \tag{22}$$

GPD is also capable of representing Gumbel, Fréchet and Weibull families as GEV does. Actually, GPD is asymptotically equivalent of GEV thus they can be alternately used [21].

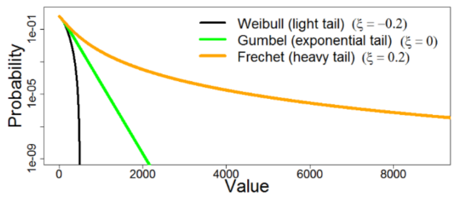
Figure 2.3a shows typical Probability Distribution Function (PDF) plots of GEV distributions with $\mu = 0$, $\sigma = 1$ and $\xi = -0.5, 0, 0.5$ respectively. Figure 2.3b illustrates PDF of GPD with $\sigma = 1$ and $\xi = -0.25, 0, 1$ respectively. Figure 2.3c illustrates tail shape of GEV/GPD distributions with Complementary Cumulative Distribution Function (CCDF or exceedance distribution) in logarithmic scale.



(a) PDF of GEV Distributions



(b) PDF of GPD Distributions



(c) CCDF of GEV/GPD Distributions [2]

Figure 2.3: Typical PDF and CCDF of GEV and GPD

Shape (ξ) is the most important parameter for both GEV and GPD which determines the type of distribution that best fits to data. In order to estimate distribution parameters, there are several methods, namely Maximum Likelihood Estimation (MLE) [22], Generalized Maximum Likelihood Estimation (GMLE) [23], quantile-quantile(QQ) plots [19], and L-moments [24].

According to [16], while estimating GEV and/or GPD distribution parameters L-moments outperformed MLE and GMLE in several situations. On the other hand, QQ-plots are not able to provide confidence intervals about estimated parameters. However, in order to estimate parameters only for EV distribution, which also known as a typical Gumbel distribution, MLE provides robust estimations.

2.1.2.4 Convergence

The continuous probability rank score (CRPS) is a type of converge test, which compares two distributions functions f_x and f_y, iteratively. It is used to detect whether the available sample size is enough or not according to a given threshold. CRPS metric is calculated by $\sum_{i=0}^{\infty}(f_x(i) - f_y(i))^2$ and if the result is lower than a predetermined threshold (e.g 0.01 or 0.001), it means that the result is not changing significantly when further samples are fed to the analysis, thus meaning that enough samples are assumed to be collected that fits to the model formed [25].

2.1.2.5 Tail Extension

Finally, by using the calculated parameters, the distribution of the sample tail is found. Inverse cumulative distribution function (ICDF) of the estimated probability distribution is used to calculate extreme values for the given exceedance probability. Taking p as the desired probability of exceedance (e.g 10^{-10}), $ICDF(p) = x$ gives the upper bound value with p. Assuming $p = 10^{-10}$, there is a 10^{-10} chance that there will be a higher value than x. In theory without using EVT, observing x would require 10^{10} test runs which is infeasible for most of the situations in real world. Of course this value is an estimation since the parameters calculated in Model Fitting phase is estimated with a confidence interval (e.g 95% for MLE).

2.1.2.6 Reliability Analysis

The quality of the estimated distribution with EVT is a challenging entity to assess. At his recent comprehensive survey about probabilistic WCET analysis, Cazorla states that no universal consensus exists to date on this issue [2]. Since the whole process is a statistical paradigm, it is reasonable to assess the reliability with some statistical tests. Some works on WCET domain [26, 27] consider the estimates obtained from the EVT are reliable only if every hypothesis of the EVT is verified. On the other hand, other approaches as in [28] propose to use standard statistical tools such as Quantile-Quantile or Mean-Excess plots to evaluate the reliability of the estimated distributions.

2.1.3 Measurement Based Probabilistic Timing Analysis

2.1.3.1 Introduction

Early studies of probabilistic timing analysis date back to 2000s [15, 29, 10, 30, 31, 9]. Burns et. al [15] is considered as the main reference in MBPTA field. They modeled the program execution time behavior with extreme value statistics and measurements, hence estimated execution times had very low probability of exceedance.

Independently of underlying assumptions and used methods, there are some principals in MBPTA which must be conformed in order to have reliable estimates. As Cazorla et al. [2] stated in his survey, the first one is the *Probabilistic Modelling* which basically seeks an assurance that both platform and the program behaves randomly which increases the possibility to observe worst case conditions during analysis runs. The second one is *Statistical Modeling* which seeks statistical evidences that the execution-time distributions conform required prerequisites. *Application Procedure* is the phase of the analysis in which the actual statistical actions are taken. The last but not the least important principal is the *Representativeness* which is a challenging and not a procedural part of the analysis. It basically seeks an evidence that the analysis observations reflect all possible cases for the operational scenarios.

Reliability of the MBPTA results are dependent on the quality of the measurements taken from the system. In practice, there are some uncontrollable disturbances on the execution times which are called as Sources of Execution Time Variability (SETV) that will be explained in the next sub-section. Therefore, there should be a mechanism that guarantees the execution time observations taken during the analysis phase upper bounds all possible operations scenarios. Figure 2.4 distinguishes the differences of scenarios.

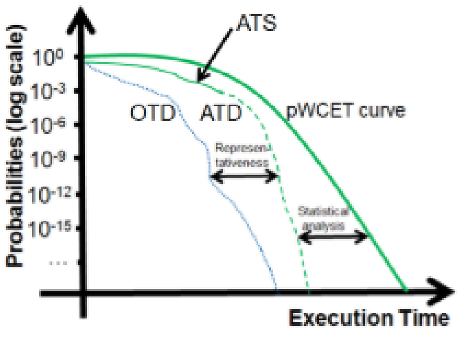


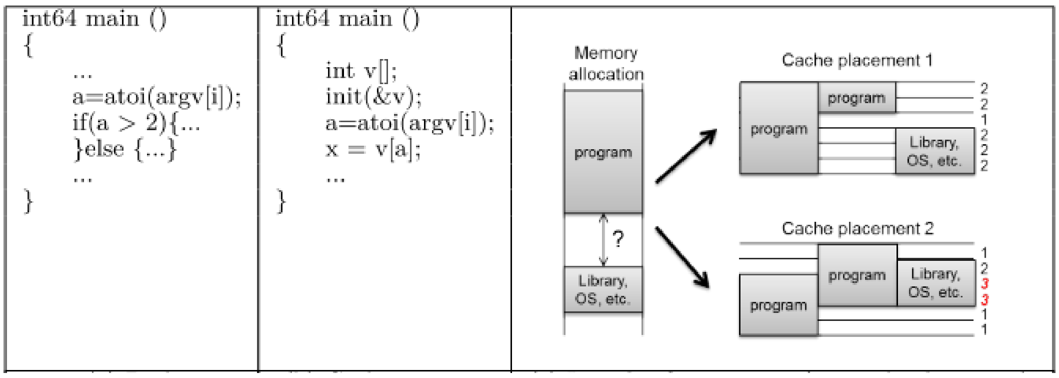
Figure 2.4: MBPTA Notions [2]

MBPTA requires analysis-time distribution (ATD) as input which should always upper bound the operation-time distribution (OTD). Aiming to this purpose is referred as *representativeness*. Applicability of MBPTA methods also require to represent the whole ATD with less samples which are referred as analysis-time samples (ATS). By using these affordable small number of samples (ATS), pWCET distribution is calculated which guarantees to upper-bound both OTD and ATD.

2.1.3.2 Source of Execution Time Variability

In complex systems, execution time of a program is affected by several conditions which are basically called as Source of Jitter (*SoJ*) [2], or Source of Execution Time Variability (*SETV*) [3]. Especially multi-path programs are input dependent so their execution time highly depends on their inputs. Those inputs do not need to be software parameters which only affects the paths exercised or floating-point operations. Additionally, hardware initial states (Cache state, I/O device state) or software initial conditions (OS Data structure, Allocation in memory) also affects the execution time.

Figure 2.5 shows some example SETV scenarios. Figure 2.5(a) represents a multi-path program which the exercised path is changed according to its input, Figure 2.5(b) shows an example of accessed memory address and whether its a cache hit or miss depends on the input value, and finally Figure 2.5(c) illustrates different object placements in cache which the latter would result in cache conflicts. Authors of [32] states that initial cache contents might decrease the overall WCET by up to 20%.



```
int64 main ()
{
      ...
      a=atoi(argv[i]);
      if(a > 2){...
      }else {...}
      ...
}
```

```
int64 main ()
{
      int v[];
      init(&v);
      a=atoi(argv[i]);
      x = v[a];
      ...
}
```

Figure 2.5: Example SETV Scenarios [3]

As Cazorla et al.[3] explains, there are several SETVs which affects directly the analysis results. Those are represented as $\{SETV_1, SETV_2, ..., SETV_j, ..., SETV_n\}$ where $j\epsilon\{1, ..n\}$ and $SETV_j = \{v^j{}_k\} = \{v_1{}^j, v_2{}^j, ...v_k{}^j\}$. For example, the execution time of a multiplication unit directly depends on its operands. $SETV_j$ for this resource can take 2^m values, where m represents the number of multiplications happening in the program.

Since we are dealing with the Worst-Case Execution Time, we are only interested in the maximum values for each $SETV_j$. However, deriving maximum values from each $SETV_j$ independently is not a trivial task. Hence, the user should provide $SETV$ which guarantees to upper bound all possible Cartesian product of maximum values of each $SETV_j$.

If $SETV_j$ would behave randomly, its effect in all $SETV$ could be ignored, thus making it independent of other $SETV$ effects. Maximum values of an independent $SETV_j$ can be selected without considering other $SETVs$. If it is not possible to make $SETV_j$ behave at random, it is necessary to make sure that $SETV_j$ works in worst conditions which guarantees to upper bound all possible $SETV_j$ effects. However, this approach results in pessimistic estimates.

On COTS platforms, making a $SETV_j$ to behave randomly is not a trivial task. According to [3], the only solution for COTS platforms is the random selection of inputs of the program and making some hardware properties to work in their worst conditions. This is referred as *Probabilistic Modelling* which makes the software program behave randomly, and hardware works in its worst conditions which serves for deriving safe ATD as was illustrated in Figure 2.4.

2.1.3.3 Application of EVT for MBPTA

The study of Cucu-Grosjean [17] is considered as the first source which details the procedural application of EVT for both single and multi-path programs to estimate pWCETs. The steps that are followed in that study coheres with the application procedure of EVT in this thesis work. However, the most significant difference between their approach and ours is that they take advantage of a randomized hardware platform to meet the i.i.d. requirements while we are obligated to build our work on a COTS platform.

Since the only solution for COTS platforms to conform the i.i.d. requirements is randomly selecting program inputs, single-path programs are not suitable to analyze with sufficient reliability. This leads us to specifically examine the WCET of multi-path programs in this thesis work. The detailed application of EVT for MBPTA in COTS platforms are given in chapter 3.

2.1.4 Copula Theory

In statistics, copulas are used to model the dependence of several random variables. A copula is basically a multivariate probability distribution with uniform marginals.

Let X_1 and X_2 be two different random variables and their cumulative distribution functions (cdf) are

$$F_1(x_1) = P[X_1 \leq x_1]$$

$$F_2(x_2) = P[X_2 \leq x_2]$$

and let H be the joint cumulative distribution function of X_1 and X_2 which is

$$H(x_1, x_2) = F_{X_1,X_2}(x_1, x_2) = P[X_1 \leq x_1, X_2 \leq x_2]$$

Here, H represents all aspects of the joint behavior of the random variables, but it is hard to interpret the dependence structure out of H. Copulas make it possible to separate the dependence structure and the behavior of the marginals described by $F_1(x_1)$ and $F_2(x_2)$. Note that, the dependence information is different from the correlation. Correlation is only one type of dependence which is a straight line between two random variables.

Let I represent the interval $[0, 1]$. A d-dimensional copula C is a cumulative distribution function on I^d with uniform marginals with a general notation

$$C(\mathbf{u}) = C(u_1, u_2, ..., u_d)$$

Sklar's theorem is the most important result regarding copulas which describes the relationship between the joint distribution H and a copula C [33].

Let F be a d-dimensional joint distribution function with marginals $F_1, ..., F_d$. Then there exists a copula C on I^d such that, for all $x_1, ..., x_d$ in $\mathbb{R} = [-\infty, \infty]$,

$$F(x_1, ..., x_d) = C(F_1(x_1), ..., F_d(x_d))$$

or

$$C(u_1, ..., u_d) = F({F_1}^{(-1)}(u_1), ..., {F_d}^{(-1)}(u_d))$$

where ${F_1}^{(-1)}, ..., {F_d}^{(-1)}$ represent the *quasi-inverse* of the marginal distribution functions.

There are also bound properties of copulas which are called the *Frechet-Hoeffding* bounds. For every copula C and (u, v) in I^2,

$$max(u + v - 1, 0) \leq C(u, v) \leq min(u, v)$$

or

$$W(u, v) \leq C(u, v) \leq M(u, v)$$

where M is the *Frechet upper bound* and W is the *Frechet lower bound.* Any copula that can be defined lays between the W and M copulas.

Extending the previous results to d-dimensional copulas results in,

$$W(u_1, ..., u_d) \leq C(u_1, ..., u_d) \leq M(u_1, ..., u_d)$$

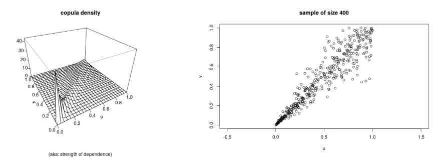
where

$$W(u_1, ..., u_d) = max \left\{ \sum_{i=1}^{n} u_i - (n - 1), 0 \right\}, u_1, ..., u_d \in I$$

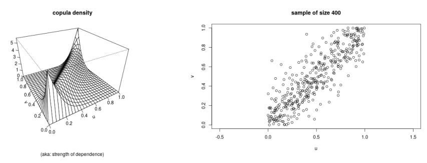
and

$$M(u_1, ..., u_d) = min\{u_1, ..., u_d\}, u_1, ..., u_d \in I$$

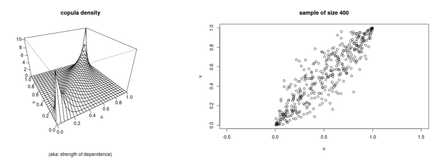
There are several type of Copulas in the literature. The most preferred ones are shown in Figure 2.6.



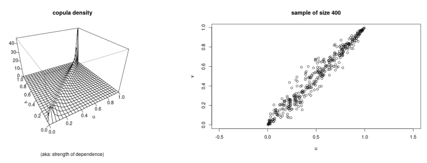
(a) Clayton Copula



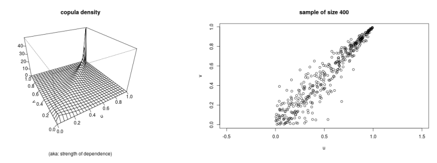
(b) Frank Copula



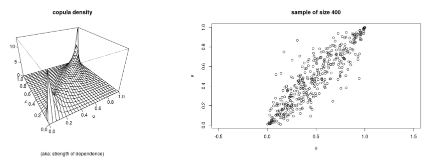
(c) Gaussian Copula



(d) Gumbel Copula



(e) Joe Copula



(f) T Copula

Figure 2.6: Widely Preferred Copulas with Their Density and Scatter Plots

The usage of copulas in order to upper bound the $Z = X + Y$ distribution dates back to 1980s [34, 35]. However, there are still recent studies on the usage of copulas in different domains such as timing analysis.

2.2 LITERATURE REVIEW

There have been numerous researches related to measurement-based timing analysis (MBTA). In this thesis we primarily focus on probabilistic variant of MBTA which is surveyed excessively in recent papers of Cazorla et al. [2] and Davis et al. [36]. Gil et al. [37] discussed the open challenges in MBPTA. MBPTA is initially studied by

Edgar and Burns [15]. They experimentally showed that it is possible to statistically model the execution time behavior of a software process running on top of a Pentium II computer with QNX real-time operation system. The input of the program is randomly generated and collected execution time samples are modeled through Extreme Value distributions [20]. They state that it is possible to predict probabilistic WCET values independent of the underlying processes and the cause of the execution time variability. However, they left the scheduling analysis out of the scope of that paper. In their other study [29], they examined the applicability of similar statistical techniques to estimate the pWCET of a task by fitting execution time observations to a Gumbel distribution. However, in that study they assumed some statistical requirements to be satisfied and used all observations to derive a Gumbel distribution instead of using *block maxima* of observations.

Hansen et al. [19] presented the use of *Block Maxima* method to estimate the pWCET through Gumbel distribution on top of a PowerPC platform with VxWorks RTOS. The estimated WCET values are validated by collecting additional millions of measurements. They showed that it is possible to safely predict the WCET values without the need of large collection of samples.

The work of Cucu-Grosjean et al. [17] is considered as the basis for MBPTA in the literature. They introduced the methodology of how to apply EVT for both single-path and multi-path programs on a simulator with random replacement cache. Execution times are collected end-to-end and applicability tests are applied in order to evidence the i.i.d. property for EVT. Samples are grouped with Block Maxima method and the Exponential Tail test is applied to check whether the distribution of maxima fits to GEV distribution. The authors point out that in order to satisfy the i.i.d. requirements for multi-path programs, it is reasonable to select the inputs randomly and group the observations sequentially.

Santinelli et al. [38] studies the effects of dependence between time observations for EVT on an Intel Xeon platform without a randomized cache or any type of bus. They experimentally evaluated that the execution time variability resulted from the underlying complex hardware architecture and is random enough for the EVT applicability. They also remark that the derived pWCET distribution is highly affected from the selected parameters such as block size for BM method or threshold for PoT method. However, they do not provide any mechanism for selecting those parameter values to achieve accurate results.

In another study of Santinelli et al. [27], they applied EVT to execution time observations taken from both time-randomized and time-predictable platforms. One of the important results of that study is that forcefully fitting distributions to Gumbel instead of determining the best fit in accordance with the shape parameter decreases the confidence of the results.

Abella et al. [14] studied the application of EVT for more realistic programs that would have step like distribution which is also called as *mixture distribution*. They argued that any real-time program must be finite and their execution time should not be heavy tailed so it is reasonable to accept the distributions to have exponential tail (i.e. Gumbel distribution). They introduced the MBPTA-CV (Coefficient of Variation) mechanism which continuously apply statistical tests to obtained samples until the exponentiality test is failed to be rejected. Finally, they fit the observed sample distribution to an exponential distribution to derive the pWCET values. This method is also converted to a tool named as *chronovise* [39]. Furthermore, Milutinovic et al. [40] discussed that the MBPTA-CV method might produce untrustworthy results when execution times are collected from different paths of a multi-path program and analyzed in a single bucket. They introduced a multiple bucket approach in which each path is analyzed on its own. Reghenzani et al. [21] discussed the misconception of forcefully fitting exponential tail distribution which supports the main idea behind MBPTA-CV.

Cazorla et al. [3] defined the general properties of MBPTA with EVT for both time-deterministic and time-randomized platforms. The authors suggest to randomize the underlying platform to increase the confidence of the results as studied later in [41]. However, they also mention that it would be possible to derive WCET estimates on time-deterministic platforms with EVT by randomly generating inputs for the multi-path programs.

There have been several case studies of MBPTA with EVT in the industry. Wartel et al. [13] applied the MBPTA procedures which are defined by Cucu-Grosjean on a case study which is designed in accordance with the Integrated Modular Avionics. The underlying platform was a PowerPC simulator with randomized cache behavior.

Silva et al. [16] studied the pWCET estimation by implementing both GEV and Gumbel fitting methods. They used L-Moments approach to estimate the parameters for GEV and MLE approach for Gumbel. Observations are grouped with Block Maxima method. They collected few samples (10^6) to estimate the low probability execution times and tightness assessments are done by comparing the results with 10^8 samples that are taken from the system. Their method is later empirically studied on an Intel platform with Linux operating system in [42].

Instead of using hardware randomized platforms for MBPTA, Curtsinger et al. [43] introduced the STABILIZER system which is also referred as dynamic software randomization in the literature. It makes use of LLVM compiler toolchain to modify the binary code during compilation and a runtime library that randomly shuffles the memory contents during the test runs to indirectly affect the cache behavior. Kosmidis et al. [44] experimentally evaluate the use of STABILIZER on a cycle-accurate simulator. Furthermore, Cros et al. [45] evaluated the applicability of dynamic software randomization with STABILIZER for MBPTA on an aerospace case study which is running on top of a LEON3 platform.

Hybrid probabilistic timing analysis (HYPTA) studies can be divided into two categories. The first one is related to the development history of RapiTime probabilistic WCET analysis tool. Years of studies performed by G.Bernat and his team are transferred into a commercial tool as detailed in [46]. They initially developed a scope-tree representation method to estimate the WCET of the programs by mitigating the effects of conventional syntax tree representation [47]. Then, they introduced the state-of-the-art biased convolution method to derive the WCET of the whole program out of the observations of small blocks [10]. It is followed by the development of pWCET tool in [30]. They also examined the use of Copulas for estimating an upper-bound on WCET [9]. That study forms the basis of this thesis. Throughout the years, the initial pWCET tool evolved into RapiTime WCET tool which is widely used in the industry [48, 49]. Law et al. [50, 51] introduced a mechanism to automatically generate test cases with search algorithms to increase the path coverage along with RapiTime. The

TACO framework introduced by Lesage et al. [52] presents a measurement framework to increase the path coverage. The authors state that the full path coverage is not possible in practice. A host computer different than the real target platform is used to simulate the program and possible input vectors are generated. Then the generated input vectors are fed into the real system which is a Rolls-Royce processor platform to collect the time samples. Those time samples are analyzed through RapiTime in the final phase.

The second group contains the studies related to the path coverage problem in HYPTA. The most dominant works are presented by Kosmidis et al. [53] and Ziccardi et al. [54] which are detailed in chapter 5.

Lesage et al. [55] presents a framework that collects measurements of small blocks from the real target, but the overall program behavior especially the followed paths are simulated on a host computer. Synthetically generated measurements are then fed to the standard MBPTA method of Cucu-Grosjean to derive the pWCET values.

Palopoli in his PhD thesis introduced a completely different approach for MBPTA by using Markov models [56].

CHAPTER 3

MEASUREMENT-BASED PROBABILISTIC TIMING ANALYSIS ON COTS PLATFORMS

3.1 Experimental Evaluation of MBPTA on a COTS Platform

In order to evaluate the proposed hybrid framework on COTS platforms, first an empirical assesment of MBPTA with EVT on our time-deterministic COTS platform should be performed.

3.1.1 Experimental Setup

The platform used in our experimental evaluation phase is a LEON3FT based ASIC platform. LEON3FT is fault-tolerant version of LEON3 SPARC V8 processor whose general architecture is presented in Figure 3.1. It is designed for embedded applications, combining high performance with low complexity and low power consumption. The CPU is running at 64MHz and it supports most of the functionality of standard LEON3 processor including the functionality to detect and correct errors in all on-chip RAM memories. LEON3FT comprise 4 set, 16k bytes/set and 32 bytes/line first level instruction (IL1) and 4 set, 16k bytes/set and 32 bytes/line data (DL1) caches. The replacement algorithm which is used in both caches is the Least-Recently-Used (LRU) replacement algorithm. The processor also implements 7-stage pipeline with Harvard architecture with the use of an efficient branch-prediction capability. It also has a high-performance, fully pipelined Floating-Point Unit (FPU). The platform also has a Memory Management Unit (MMU), but we choose to disable it since the underlying real-time operating system (RTEMS RTOS) does not have virtual address translation capability, thus does not support MMU.

LEON3 platforms have a Debug Support Unit (DSU) which is basically a debug module for the processor and CPU Advanced High-performance bus (AHB). To simplify debugging on target hardware, LEON3 processor implements a debug mode during which the pipeline is idle and the processor is controlled by the DSU. It enables to control processor execution (hardware breakpoints, single stepping, etc.) and to access the registers and the caches of the processor. DSU also provides a trace buffer for the instructions and a trace buffer for the CPU AHB transfers. However this trace buffer has only 8k bytes length which is not suitable for long benchmark runs to collect execution time traces for the whole program. The underlying platform has 256MB on-chip RAM along with the LEON3 CPU which is used to store execution time traces initially for this work.

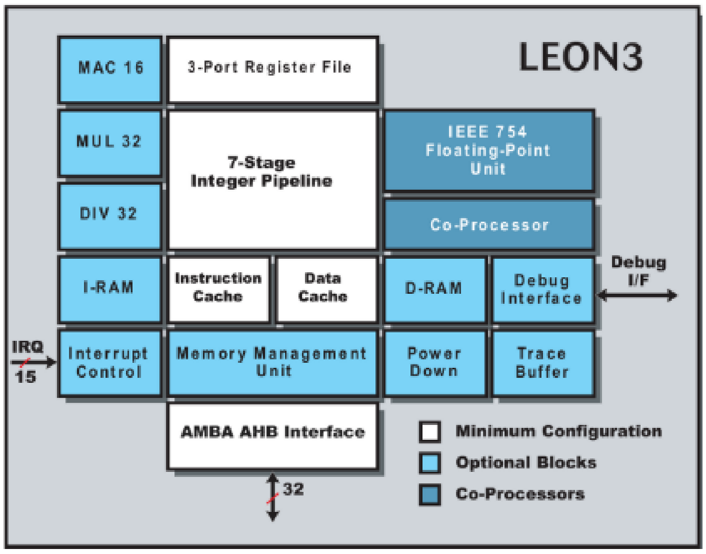


Figure 3.1: LEON3 Architectural Blocks

The software side of the platform comprises a real-time operating system (RTOS) named as RTEMS. The version that we use in our work is a commercially certifiable version. RTEMS is a widely used RTOS mainly in space flight systems, and also medical and embedded network systems. In architectural point of view, RTEMS does not support virtual memory management and processes, but it implements a single process in a multithreaded environment.

Higher layers of the software environment is constructed in accordance with Integrated Modular Avionics (IMA) [57, 58] concept which basically divides our so-called processes into partitions temporally and spatially. Since RTEMS does not support virtual memory management, spatial partitioning is not possible for now. There are some ongoing works about implementing ARINC-653 partitioning concept onto RTEMS [59]. However, this requires altering the RTOS source code which is not possible in our case due to certification considerations. Temporal partitioning is done under the priority based preemptive scheduling concept. A non-preemptive scheduler task partition, which has the highest priority, schedules other so-called processes (preemptive tasks) with a timer. Each so-called process includes components which are cyclically scheduled inside assigned partition duration. Figure 3.2 represents aforementioned scheduling concept of our running environment. For the simplicity of experimental evaluation, all partitions except the analyzed one are disabled.

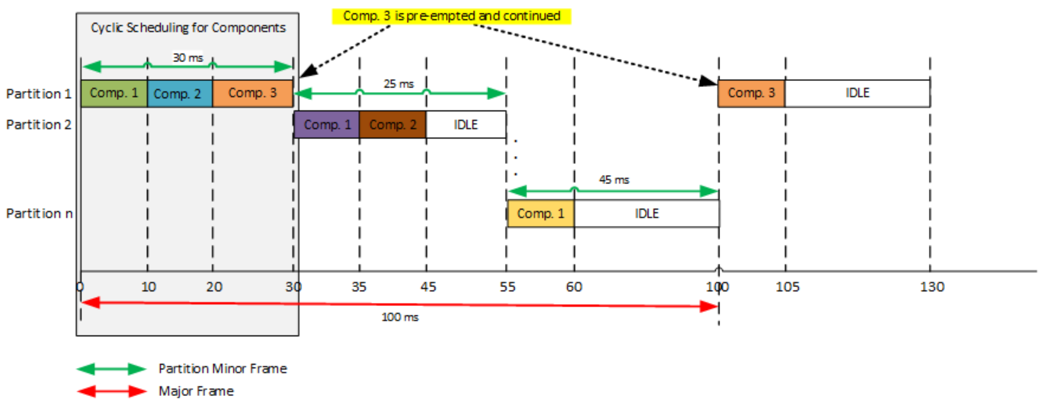


Figure 3.2: Partition Based Scheduling of Components

One of the main purposes of partitioning is to prevent a failure in one partition should not propagate to the other in timely manner. While this ensures that each partition will start its duty exactly on-time, it does not consider the timing requirements of software components inside the partitions. Because of that, processor utilization is one of the main challenges in partitioned systems. In order to comply with the timing requirements of the functional behavior of system and increase the utilization, software components inside the partitions should be distributed carefully. Hence, the main factor to analyze to overcome this problem is the WCETs of the components under question [60].

In order to evaluate and experiment the applicability of MBPTA with EVT in our time-deterministic COTS platform, several benchmark programs are structurally translated into components. These components are compiled using RTEMS LEON/ERC32 GNU cross-compiler system (RCC) [61], which basically compiles the program with GNU GCC and links with RTEMS RTOS which finally generates *.ELF* object file to be programmed into the target hardware. Some of the compiler flags used are *-O0* which is to explicitly force to compile without any optimization, and *-msoft-float* to emulate floating-point operations without any hardware support.

Final representation of our experimental program is shown in Figure 3.3 which is highly representative of an actual time-critical program running on top of a COTS platform. For simplicity, all partitions and irrelevant components are disabled, but underlying layers are left as is in order to reflect the real system as much as possible.

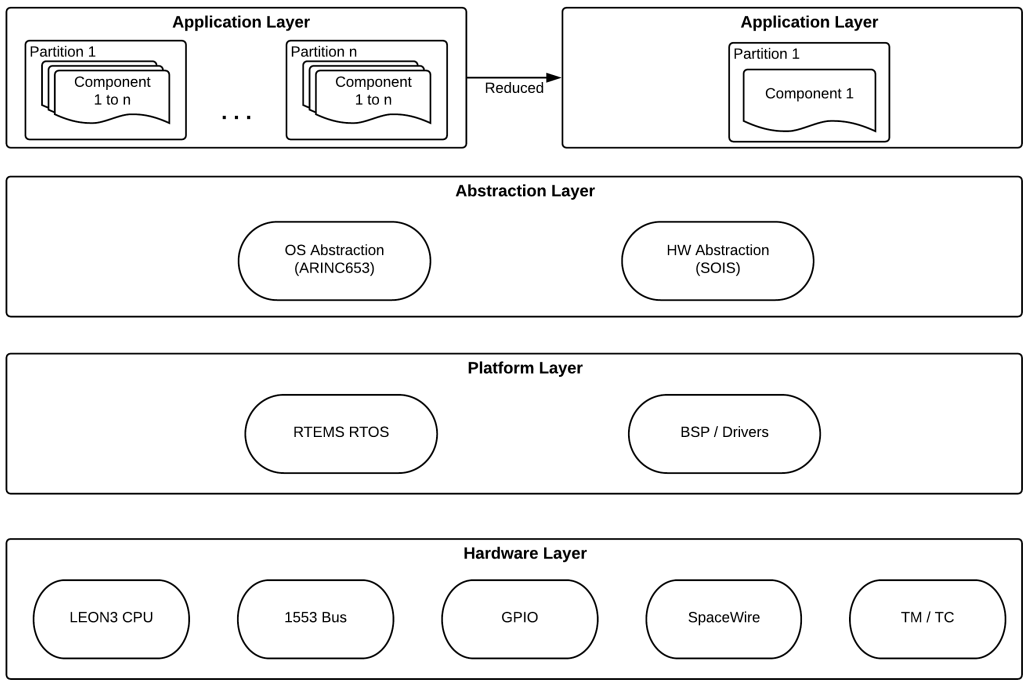


Figure 3.3: Architecture of Experimental Benchmark Program

3.1.2 Definition of Inputs

As stated before, the population under analysis is the execution time traces drawn from software components in our system. Before going any further, we need to define what a component is and how its behavior affects the timing analysis procedure.

Components are composed of several sub-functions which collaboratively operates in response to the requested function. Their input points and the input definitions are strictly defined, hence if those inputs do not satisfy the prerequisites, they are basically rejected and that component does not do anything on its own. This is important since it means that components are highly input-dependent and more importantly they are multi-path programs as they are composed of several sub-functions.

In order to experimentally evaluate the conformance of MBPTA on our COTS platform, we chose to implement some of the Mälardalen WCET Benchmarks [62]. The main drawback of these benchmarks is that their predefined WCET values are only valid for static analysis approaches and also their input ranges are not given. Because of these reasons, we carefully examined some of the programs and derived some bounds on the input parameters to generate random input vectors. In fact, deriving very accurate parameter bounds are not necessary since we are only dealing with the execution time behavior of the program and we assume that those programs might accept those inputs and derived execution times are accepted as long as the programs do not unexpectedly terminate.

In our EVT experiments we chose to use the following benchmark programs since they are highly input-dependent and performs floating-point operations on arrays which makes them sensitive to hardware effects:

1. *fir* = Finite Impulse Response filter program which has a lot of vector multiplications and array handling. Its structure is composed of nested loops in which the inner loops depend on the outer loops.

2. *select* = Selection of Nth largest element from a given floating point array which has a lot of floating point operations. Its structure is composed of 3-level nested loops.

3. *janne_complex* = It is a nested loop program in which the bounds of the loops are dependent on the input and inner loops depend on outer loops.

3.1.3 Applying EVT on Time-Deterministic COTS Platform

The stages of experimental evaluation are depicted in Algorithm 1. Each benchmark program under analysis is translated into a component as explained above. The component is encapsulated with benchmarking code instrumentations to randomize inputs to make the program traverse different paths and to obtain end-to-end measurements which is explained as Algorithm 1. Encapsulated components are compiled using RCC toolchain which is finally linked with RTEMS RTOS. Test runs are performed until they finish, instrumented components are tested for a predefined number of times i.e 10000 for this experiment.

Algorithm 1 Testbench for MBPTA Trace Capturing

```
1: $buffer \leftarrow$ ALLOCATEMEMORY($for$ $10000$ $traces$)
2: function TESTBENCH
3:     for $i = 1$ to $10000$ do
4:         $x \leftarrow$ GENERATERANDOMINPUT
5:         $buffer \leftarrow$ GETCURRENTTIME
6:         FLUSHCACHES
7:         COMPONENT($x$)
8:         $buffer \leftarrow$ GETCURRENTTIME $-$ $buffer$
                                        ▷ Store the end-to-end time difference
9:     end for
                          ▷ $buffer$ is dumped through GDB interface at this point
10: end function
11: function COMPONENT($x$)
                                             ▷ Do something that depends on $x$
12: end function
```

In order to randomize any property in software, a reference random number is needed. Hence, to increase the confidence of randomization process, *Mersenne Twister* pseudo-random number generation method is implemented since it has been proven to be "more random" than the built-in generators in many common programming languages including C [63].

Before each run, inputs of the components are shuffled or randomly generated. Then, caches are flushed as explained in [64] and current time is fetched from RTOS which basically gets the current time value from a timer in nanoseconds resolution.

Then, the component is executed by using randomized inputs and current time is fetched once again and finally time difference is stored inside an allocated RAM area, which is large enough to store 10000 execution time traces.

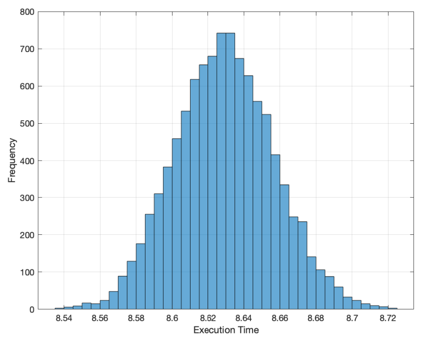
When the benchmark run is finished, stored traces inside the RAM area is dumped onto the host computer by using DSU interface. In order to use DSU interface, the running program must be stopped and the processor should be switched to debug mode as was explained in Experimental Setup subsection.

Once end-to-end measurements for a component are obtained, implementation of EVT starts, which is detailed in the subsections below.

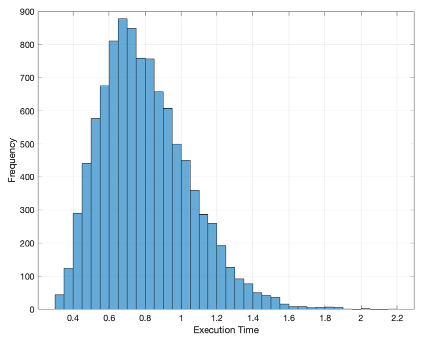
3.1.3.1 Initial Observations

Benchmark programs are analyzed and 10000 observations are collected separately. Figure 3.4 shows the histogram plot of collected execution times for each benchmark program.

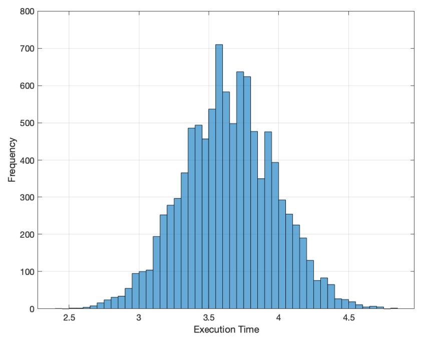
In Figure 3.4, *fir* looks normally distributed while *select* tends to have heavy tail. *janne_complex* also looks like normally distributed with unusual spikes. However, we are not interested in their overall shape. Since we are dealing with Worst-Case Execution Time, we only need a model for their tail behavior which are the values that are laid on the rightmost parts of the plots. EVT models each tail of distribution and it does not seek for a good fit for its internals.



(a) fir



(b) select



(c) janne_complex

Figure 3.4: Histogram of Benchmark Programs

Table 3.1 below summarizes the initial statistics of the observations.

Table 3.1: Execution Time Statistics of the Observations

Program	$Minimum$	$Maximum$	$StdDeviation$
fir	8.5357 ms	8.722 435 ms	0.027 478 5 ms
select	0.318 65 ms	2.152 825 ms	0.246 46 ms
janne_complex	2.434 315 ms	4.834 29 ms	0.331 575 ms

3.1.3.2 Applicability Evidence

EVT requires the samples to be *i.i.d* and in order to verify that assumption several statistical tests must be performed. Independence means that each sample that is observed during the analysis is not affected by another measurement. For providing evidence of independence we used Wald-Wolfowitz (WW) and Ljung-Box (LB) statistical tests.

WW, also called as *runs* test, has the null hypothesis of randomness of the observations. It marks the observed values as + and - which are greater and lower than median, respectively. Values that are equal to the median are discarded. It then examines the normality of the runs (continuous + and - marks) [42].

LB or *Box* test has the null hypothesis that there are no serial autocorrelation between observations up to specified lag. There are very little practical advice on how to choose the number of lags so we have calculated for several lags as in line with Silva et. al [42].

Identicality evidence can be provided with Two-Sample Kolmogorov-Smirnov (KS) and the k-sample Anderson-Darling (AD) statistical tests.

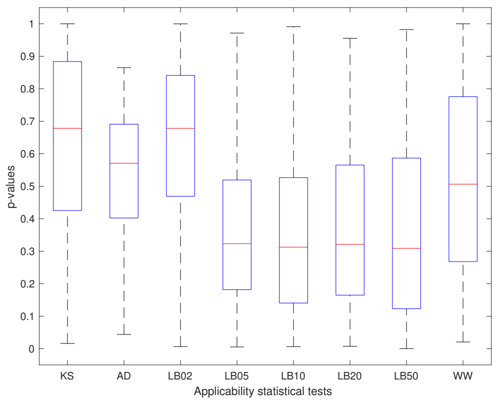
KS test is a nonparametric hypothesis test that evaluates the difference between the cdfs of the distributions of the two sample data vectors. It has the null hypothesis that the samples in each vector are from the same distribution.

k-sample AD test is also a nonparametric test that checks whether given two or more groups of data are identical. Each group should be randomly drawn from a population. It is introduced by Scholz and Stephens [65] and is a generalization of Two-Sample AD Test. It is also a modification of KS test which gives more weight to the tails.

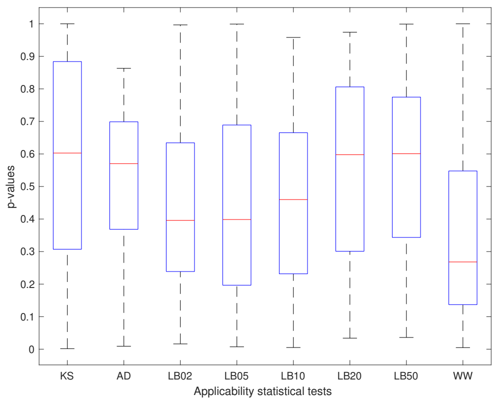
In order to increase the credibility of these tests, using our 10000 observations, we formed two randomly selected 1000 sample blocks. After obtaining two groups in which each of them contains 1000 consecutive observations, they are tested by using the aforementioned statistical tests. This process is repeated 1000 times, which means we created two random blocks for 1000 times and obtained 1000 p-values for each statistical test. Obtained p-values should be uniformly distributed in the range $[0, 1)$ and tend to be high in order to pass the tests.

Results are presented in Figure 3.5 by using whisker-box plots with the 0%, 5%, 50%, 95%, 100% quantiles which are minimum, 5%, median, 95% and maximum of the obtained p-values, respectively. They are much higher than 0.05 and uniformly distributed which provides enough evidence that the observations conform the *i.i.d* requirement in order to apply the EVT.

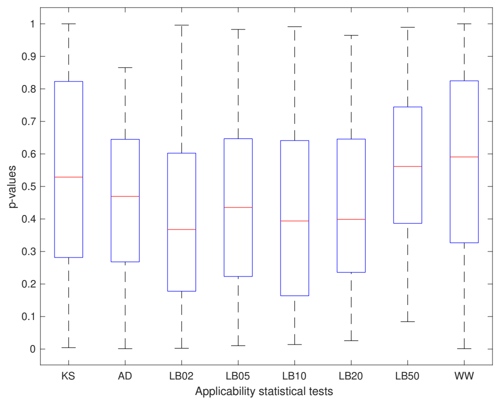
All the tests are implemented in MATLAB environment and the codes are given in Appendix A.1.



(a) fir



(b) select



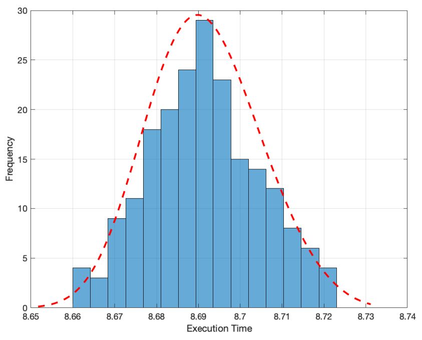
(c) janne_complex

Figure 3.5: Statistical Tests of Benchmark Programs

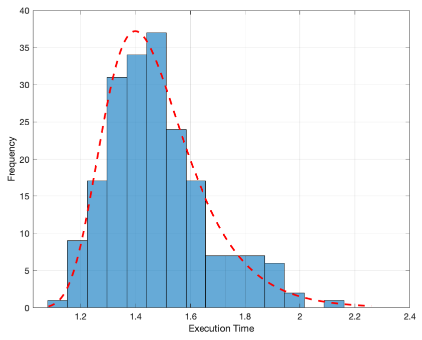
3.1.3.3 pWCET Estimation

Obtained observations represents the ATD which was explained in Figure 2.4. In order to represent the whole ATD with less samples (ATS), a data selection process should be performed. We are interested in the WCET which means that we need to specially model the rightmost tail behavior of the ATD. As was explained before, there are two main approaches in EVT to filter out the tail, which are BM and PoT. Depending on the selected filtering method, obtained ATS either fits to GEV or GPD, which we already noted that they can be used alternately. We followed the similar approach as [17] while applying the EVT steps for our COTS platform.

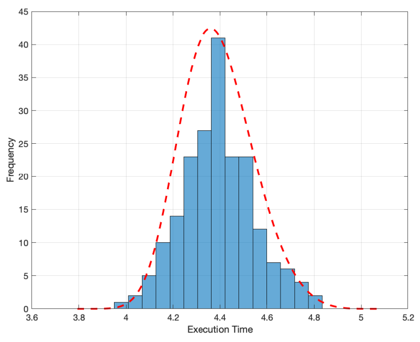
For evaluation purposes, we have selected to use BM approach in this section. Observations are divided into blocks of 50 samples as proposed in [17] and maximums of each block are selected to construct ATS of each ATD. Figure 3.6 shows the histogram of resulting distributions of each ATS.



(a) fir



(b) select



(c) janne_complex

Figure 3.6: Filtered Samples with BM Approach and Fitted GEV PDF Plots

After obtaining ATS, the next step is to estimate the model parameters. Because we used BM approach, ATS should follow a GEV distribution. In our thesis we used L-Moments approach [24] in order to estimate the model parameters. The estimated

GEV parameters for each program are given in Table 3.2. Furthermore, by using those parameters, fitted GEV PDFs are plotted on Figure 3.6. The MATLAB codes to get block maxima of ATDs and parameter estimations with L-Moments and MLE approach are given in the Appendix A.2, A.3 and A.4.

Table 3.2: Estimated GEV Parameters

Program	$Shape(\xi)$	$Scale(\sigma)$	$Location(\mu)$
fir	-0.2339	0.0130	8.6864
select	-0.0119	0.1463	1.3970
janne_complex	-0.1962	0.1452	4.3295

It is worth noting that the shape $\xi \leq 0$ for all distributions. It means that they all fit to a Weibull distribution which has light tail with a sharp slope and has a maximum value. We mentioned before that *select* tends to fit to a Fréchet distribution, but its tail behavior follows a Weibull and almost Gumbel behavior.

EVT aims at deriving a distribution which safely upper-bounds the tails of the observation distributions. Silva et al. [16] empirically showed that Gumbel is safer than GEV by providing reliability and tightness results. He states that Gumbel should be used for timing analysis instead of GEV. Moreover, Abella et al. [14] clearly explains why Gumbel is more suitable for timing analysis by identifying some constraints as follows:

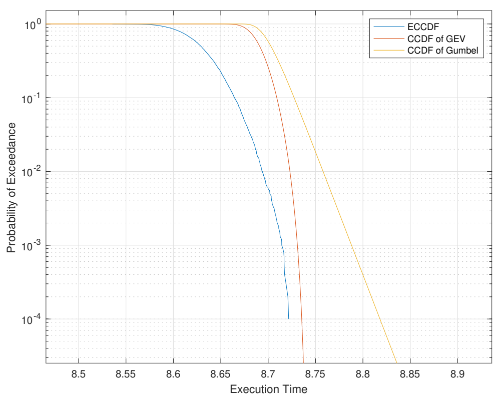
1. A program is naturally finite which means that it has some code constraints such as loop bounds. Especially in a time-critical domain, those constraints are strictly defined and programs are constructed by limited number of instructions which have maximum execution times. Because of these reasons, authors of [14] state that execution time of a time-critical program should not be heavy tailed (Fréchet). Instead, they should have a maximum execution time.
2. EVT seeks a safe upper-bound and it should prefer pessimism over optimism when needed. In a time-critical domain, if increasing accuracy also increases the possibility of optimism, it should stay in the pessimistic side in order not to lose safeness.

Abella et al. [14] also propose a method to estimate the parameters for Gumbel distribution which is basically based on collecting as much observations as possible until it satisfies the necessary requirements. They state that any time-critical program should converge to a Gumbel distribution eventually if enough observations are taken and they check that by applying Exponential Test (ET) [66]. Once ET is satisfied, they estimate the parameters μ and σ while enforcing $\xi = 0$.

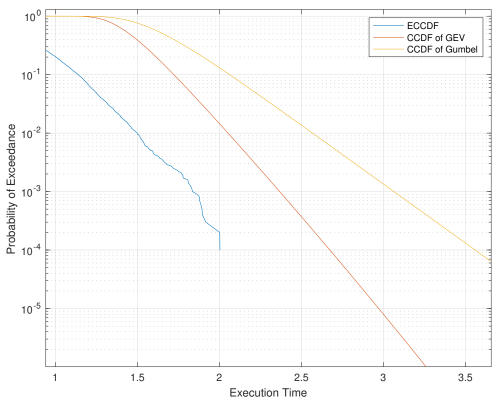
In our evaluation, we did not follow the same approach. In fact, we assume that we have collected "enough" observations based on the GEV fitting results given in Table 3.2 which shows that all distributions are light tailed. It means that if we enforce any of those distributions to follow a Gumbel model, it will definitely upper-bound our GEV results. However, this assumption may result in lack of evidence for a good model fitting, which is explained in the next section in detail.

As we have already calculated the parameters for GEV model. Estimation of Gumbel parameters is best done with MLE approach [16] so we also calculated the parameter values for Gumbel distribution. The Figure 3.7 shows the resulting Empirical Complementary CDF (ECCDF), CCDF of GEV and CCDF of Gumbel respectively in logarithmic scale in order to examine the tail behavior in details. Complementary CDF shows the probability of exceedance for each estimated WCET value.

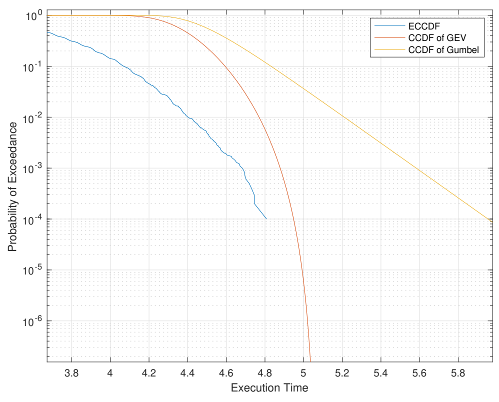
Blue line represents the empirical CCDF which is a non-parametric distribution and shows the real sample distribution. Red line represents the CCDF of GEV model which both with Gumbel upper-bounds the whole ECCDF which is the expected behavior since the GEV and Gumbel is constructed by using the tail values of the real samples.



(a) fir



(b) select



(c) janne_complex

Figure 3.7: Complementary CDF Plots in Logarithmic Scale

Table 3.3 summarizes several pWCET estimations for each program. The calculation is done by using inverse CDF functions of each distribution by using the desired probability of exceedance parameter.

The MATLAB codes to derive the results in the table are given in Appendix A.6.

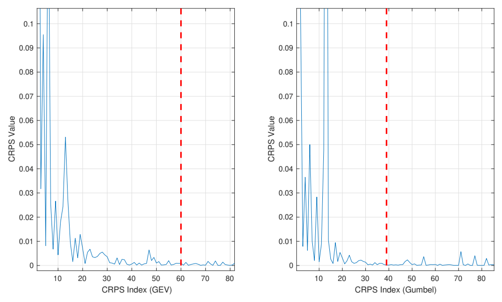
Table 3.3: pWCET Estimates

Program	*fir*	*select*	*janne_complex*
HOET	$8.7224\,\mathrm{ms}$	$2.1528\,\mathrm{ms}$	$4.8343\,\mathrm{ms}$
GEV(10^{-4})	$8.7355\,\mathrm{ms}$ (0.1%)	$2.6731\,\mathrm{ms}$ (24%)	$4.9479\,\mathrm{ms}$ (2.3%)
GEV(10^{-6})	$8.7397\,\mathrm{ms}$ (0.2%)	$3.2604\,\mathrm{ms}$ (51%)	$5.0202\,\mathrm{ms}$ (3.8%)
GEV(10^{-9})	$8.7415\,\mathrm{ms}$ (0.2%)	$4.0829\,\mathrm{ms}$ (89%)	$5.0567\,\mathrm{ms}$ (4.6%)
Gumbel(10^{-4})	$8.8179\,\mathrm{ms}$ (1%)	$3.5591\,\mathrm{ms}$ (65%)	$5.9541\,\mathrm{ms}$ (23%)
Gumbel(10^{-6})	$8.8779\,\mathrm{ms}$ (1.7%)	$4.5499\,\mathrm{ms}$ (111%)	$6.6971\,\mathrm{ms}$ (38%)
Gumbel(10^{-9})	$8.9677\,\mathrm{ms}$ (2.8%)	$6.0360\,\mathrm{ms}$ (180%)	$7.8117\,\mathrm{ms}$ (61%)

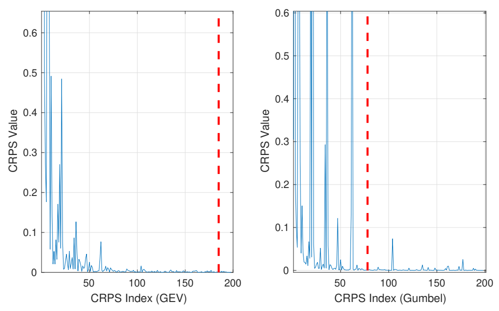
10^{-4}, 10^{-6} and 10^{-9} probability parameters correspond to the probability of the execution time of the program exceeds the estimated value. Result of pWCET(10^{-6}) states that out of 10^6 runs of the program, observations should not exceed the calculated value.

From the table above, results of *fir* does not change significantly that is because the *fir* program has bounded loops which strictly limits the possibility of executing longer than already observed values. The variations come from the floating point operations and data accesses during the filtering phase. Even in the worst case which is Gumbel (10^{-9}), calculated pWCET is only 3% higher than HOET. *select* on the other hand shows almost an unbounded behavior. It has a 3-level non-rectangular loop which has a lot of floating point comparisons and most importantly the loop bounds are not strictly defined which is iterated until it finds a proper solution. Capturing more observations would decrease its variable tail behavior, thus might have a sharper slope. Lastly, *janne_complex* has the most variation among others for its ATD. That is because it has nested loops whose bounds are directly dependent on the given input. Since we have limited the input value range during the random generation phase, its execution time is bounded from a higher level.

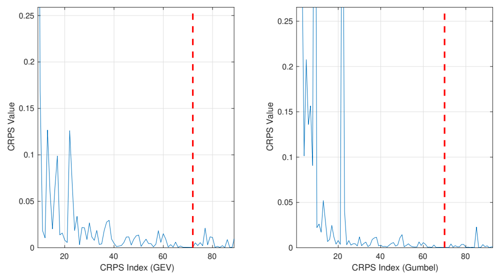
Another important step in measurement-based timing analysis is to decide when to stop capturing traces. One of the possible numeric techniques is the CRPS test. In our evaluation we calculated the CRPS values by using both GEV and Gumbel models and we set the acceptance threshold as 0.001. As in line with Cucu-Grosjean et al. [17], in order to eliminate the possible false positive cases, we set the convergence round limit as 5 which means that the calculated CRPS should be below the threshold for 5 consecutive rounds. In each round, we increased the sample size by 50 starting from 100 samples until 10000 which totals to 198 rounds and at each round the data sampling with BM method and model fitting is done as explained before. Finally, CRPS value is calculated for each round and checked whether the convergence criteria is met or not. Figure 3.8 shows the results for each program under analysis for both GEV and Gumbel models. Blue line plot represents the CRPS change, red dashed line represents the index value where CRPS test has converged.



(a) fir



(b) select



(c) janne_complex

Figure 3.8: CRPS Plots for Both GEV and Gumbel Models

According to the results of CRPS test, it would be enough obtain 3100 (GEV) and 2050 (Gumbel) samples for *fir*, 9350 (GEV) and 4000 (Gumbel) samples for *select* and 3700 (GEV) and 3600 (Gumbel) samples for *janne_complex*. It can be seen from the figures that after some point, CRPS values do not change significantly which means that collecting more values would not affect the fitted model.

The MATLAB codes to calculate the CRPS values are given in Appendix A.5.

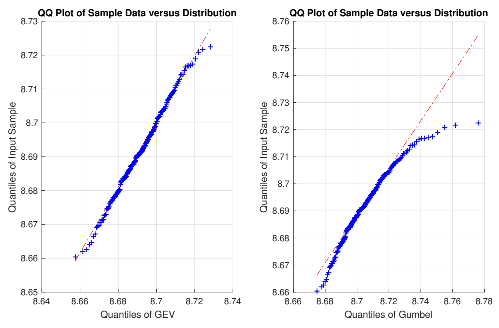
3.2 Reliability and Tightness of the Results

So far, we have applied all the steps of EVT for MBPTA as in line with Cucu-Grosjean et al. [17] that is considered as one of the pioneers which describes step by step procedures of applying EVT for both single-path and multi-path programs.

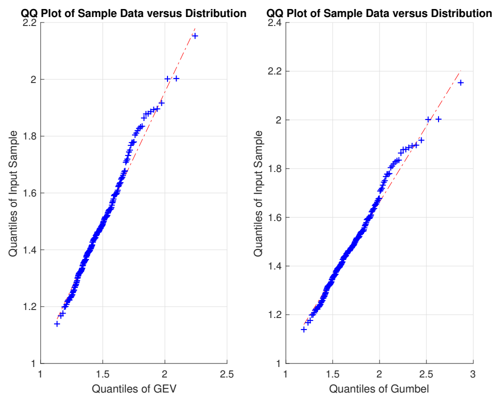
Reliability and tightness of the results is one of the important questions of EVT. Most of the works in this domain assume that the derived results are reliable and tight. Cazorla et al. [2] states that there are no universal consensus to date on how to asses the reliability of the derived results from EVT. However, there are several methods which are assuming the results are reliable indirectly by trusting the confidence levels obtained from the applicability statistical tests, or Quantile-to-Quantile plots or the Mean-Excess plots.

Tightness is another issue since the actual WCET is not known, it is not possible to derive how tight is the estimated distribution. Silva et al. [16] proposes a solution which takes advantage of collecting huge number of observations from the system and marks the highest observed execution time (HOET) and compares that value with the estimated pWCET result which is derived by using smaller samples.

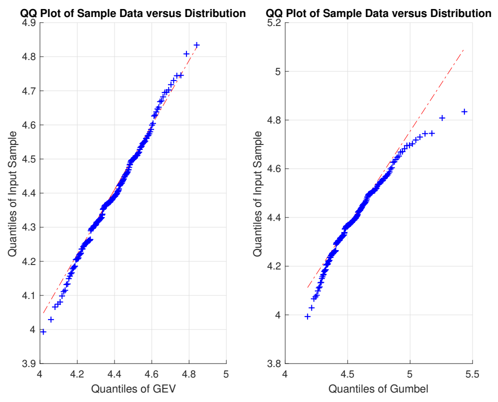
In this evaluation work, we chose to implement QQ-plot approach for reliability assessment and huge number of sample collection approach for tightness assessment. QQ-plot is a plot which shows the quantiles of the sample data versus quantiles of the desired distribution. If the samples fit to the distribution, the plot produces an approximately straight line, suggesting that the sample data and the desired distribution have the same behavior.



(a) fir



(b) select



(c) janne_complex

Figure 3.9: QQ Plots for Both GEV and Gumbel Models

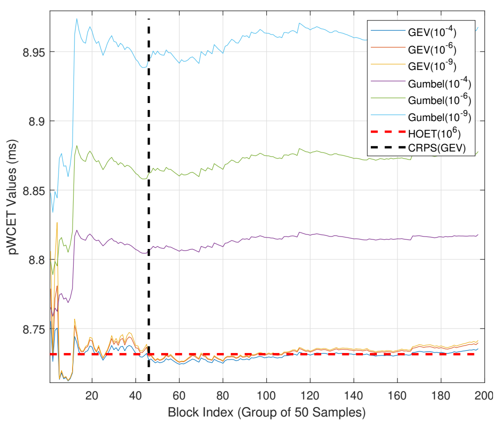
Figure 3.9 above shows that GEV models fit better than Gumbel to our samples. However, we already mentioned in the previous section that enforcing the data to fit to Gumbel would result in lack of good-fit to the model which is illustrated here. This is because Gumbel stays in the pessimistic side in order to not to lose safety cause, but its tightness is questionable. On the other hand, GEV is more vulnerable to possible unusual outcomes of the programs in the future, but provides tight results.

An important reminder is that the results which are derived from EVT is only valid for the conditions which the analysis runs are taken. Since we do not know the input which would let the program traverse the worst-case path, randomly generated inputs are given to the program. Hence, the results are derived by using randomly selected inputs and hardware variations due to having huge number of floating-point operations and several data access patterns. There are also other hardware effects covered which is not possible to independently mention meaning that the resulting pWCET distribution is only valid for the given program inputs and tested hardware conditions.

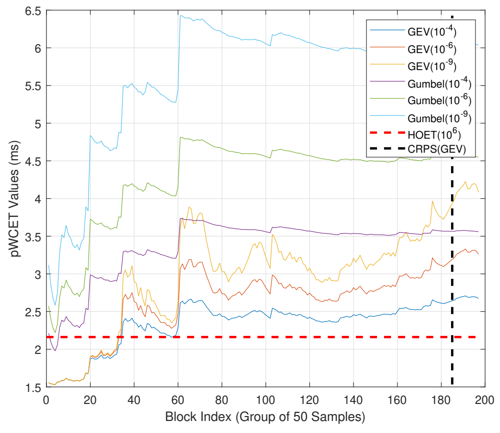
Although we have limited control over the underlying COTS hardware, it is possible to control the inputs. In order to provide an insight for how tight and reliable the results are, we have also stored the randomly generated inputs for each program during the analysis phase. It means that we have stored 10000 inputs for each program. By doing that we have extracted which input parameter resulted in the $HOET$ during the analysis. Hence, we now have $HOET(10^4)$, $pWCET_{GEV}(10^{-4})$, $pWCET_{GEV}(10^{-6})$, $pWCET_{GEV}(10^{-9})$, $pWCET_{Gumbel}(10^{-4})$, $pWCET_{Gumbel}(10^{-6})$, $pWCET_{Gumbel}(10^{-9})$ and $Input_{HOET(10^4)}$ for each program to be used during reliability and tightness analysis.

In order to compare the results, we modified the input generation phase in Algorithm 1, and assigned the same $Input_{HOET(10^4)}$ which is derived during the analysis phase for all the iterations. More specifically, we have given the same $Input_{HOET(10^4)}$ parameter 10^6 times to the program. We followed this approach because if we would continue to generate random inputs, it would be meaningless since it would definitely generate different inputs other than our tested inputs, which because EVT does not guarantee providing a safe upper bound for such a case. Coverage is another problem for Measurement-Based timing analysis, which is covered in Ch. 5 in detail.

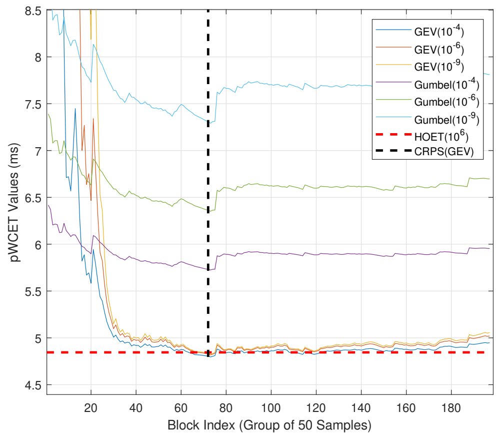
Finally, we have collected 10^6 samples for each benchmark program with the same given input for comparison. Figure 3.10 shows both the GEV and Gumbel CDFs and $Input_{HOET(10^6)}$ which is obtained from 10^6 runs.



(a) fir



(b) select



(c) janne_complex

Figure 3.10: pWCET Reliability and Tightness Analysis Results

Red dashed line represents the obtained $HOET(10^6)$ and black dashed lines are the $CRPS$ indexes. It is clearly seen from the figures that GEV models are much tighter than Gumbel. However, for the *fir* case GEV is so tight that it sometimes underestimates the WCET. Especially when $CRPS$ is trusted GEV totally underestimated the WCET which also shows that $CRPS$ is not a highly trustable metric. However, after the $180th$ index which corresponds to 9000 samples, GEV tends to stay above the $HOET(10^6)$. For the *select* case, GEV seems to be both tight and reliable since it stays above the $HOET(10^6)$ after the $60th$ index which corresponds to 3000 samples. In all cases, $pWCET_{Gumbel}(10^{-4})$ seems to be both tight and reliable.

3.3 Conclusion

In this chapter, we provided a systematic way to assess Measurement-Based Probabilistic Timing Analysis (MBPTA) with EVT on COTS platforms. Almost all the studies in literature focuses on randomizing the hardware in order to increase the reliability and representativeness of the samples. However, there are very limited studies for COTS platforms.

In reality, COTS platforms do not allow for cache policy or bus arbitration modifications. It is also not possible to obtain measurements in instruction granularity from COTS platforms as it is possible for simulators or specialized hardware which is to be studied in chapter 4.

Along with all these limiting factors, we empirically assess the applicability and reliability of MBPTA with EVT for some previously selected WCET benchmark programs. Those programs are specifically selected to get hardware effects involved. Non-rectangular loops, input dependent data accesses, input dependent conditional paths and loop bounds made each program to depend heavily on hardware features and the given software input.

Studies in literature assume that the worst-case path input is already known and by giving that input to the program they basically eliminate the path effect from $SETVs$. However, in reality it is not possible to derive that input which is why there are studies in order to increase the path coverage in measurement-based timing analysis domain. In our EVT assessment we have generated random inputs for the test programs which is a realistic approach to increase path coverage and generate necessary variations in execution time. Those variations result from the fact that the benchmark programs are multi-path, which means that they are input dependent. We also selected programs, which do a lot of complex operations on data in order to be realistic.

We defined the component term in order to explain that in reality program inputs are strictly defined and their value ranges are known from the design. It makes us to generate random inputs in accordance with those range constraints.

While applying EVT for MBPTA of our benchmark components we simply followed the following steps:

1. Collect 10^4 measurements and store the generated random inputs for each program.
2. Apply statistical tests to the ATD values.
3. Derive ATS from ATD by applying BM method.
4. Estimate the GEV parameters μ, σ and ξ for each ATS.
5. Forcefully fit ATS to Gumbel and estimate the parameters μ and σ while enforcing $\xi = 0$.
6. Derive the CRPS index.
7. Plot the QQ-Plots to check goodness-of-fit of the models.
8. Collect 10^6 measurements with $Input_{HOET(10^4)}$ for each program and compare the pWCET results with $HOET(10^6)$.

Some of those steps are done only for experimental evaluation purposes. In reality, only the $1st$, $3rd$ and $4th$ or $5th$ steps are sufficient to estimate the desired pWCET value with a desired probability of exceedance. We showed that the $CRPS$ index is not that trustworthy since it may result in underestimations as was shown in Figure 3.10. Furthermore, QQ-plots do not provide numerical results so it may only be used for visual verification purposes. Finally, collecting 10^6 measurements just for checking would not be possible for some realistic programs so that step can also be omitted.

One important point to consider is that all measurements are taken end-to-end in here. In reality, programs are not that small as our benchmark programs. On the contrary the real tasks or functional components are composed of several sub-programs. It is still possible to apply EVT for such cases, but the more the program complexity increases, the more its path coverage through randomized inputs are decreased. Therefore, Hybrid Probabilistic Measurement Based Timing Analysis (HYPTA) methods have emerged which combines some properties of Static Analysis domain and some properties of MBPTA. Thus, HYPTA seeks to increase path coverage based on static properties of the program and measurements obtained for program blocks. Of course it also has some drawbacks that needs to be studied, which is the main motivation and direction of this thesis. By using and assuming some of the outputs of the works done in the present chapter, we present our method in chapter 5 in detail.

CHAPTER 4

INCREASING CONFIDENCE OF MEASUREMENTS

Collection of measurements from a computing system can be done in several ways. A simple approach is to instrument the source code to collect time stamps or get CPU cycle counter values directly from the system. Another widely used method is to use external hardware such as logic analyzers to capture and time stamp special signals from the system. There are also non-intrusive ways which require customized hardware tracing mechanisms such as NEXUS standard [67] or Embedded Trace Macrocell (ETM) mechanism from ARM [68]. Using simulators to extract measurements are another non-intrusive method to collect measurements.

Non-intrusive mechanisms require customized hardware support which are not widely encountered in commercial products. The most commonly used method is injecting instrumentation points, which are triggered when the program is run to emit an identifier. That identifier is then time stamped to construct the timing trace for the program segment.

As A.Betts explained in his PhD thesis [69], software only solutions time stamp trace points on the target and stores the traces inside internal memory. As the memory space is very limited on embedded systems, this method provides no scalability. Hence injected instrumentation code produces timing penalty and increases overall code size, which is commonly referred as **probe effect**.

The main drawback of Measurement Based Timing Analysis is the probe effect generated in collecting measurements. There have been several studies to eliminate or minimize the probe effect to derive more confident results. In [70], Betts et al. collect time stamped traces of objects non-intrusively through special hardware via Nexus debug interface which is limited by the size of debug buffer to store huge amount of trace data. Karlsson [71] in his MSc thesis, developed an FPGA based trace mechanism which benefits from open-source SPARC IP-library. Traces are generated on board and through co-operating SoC on the same board, they are transmitted to a computer on PCI bus. Dreyer et al. in [72] proposes an FPGA architecture to continuously collect traces non-intrusively from an Embedded Trace Unit which is again a specialized hardware property. Commercial Hybrid Measurement Based Analysis solutions such as RapiTime [49] along with the company's commercial data logging solution RTBx provides a measurement-based WCET analysis platform. Traces are collected with aforementioned software-only solutions or software/hardware mixed solutions through RTBx.

In this thesis work, we also implement a mixed software/hardware method similar to the RTBx product of Rapita Inc. The main driving reasons why we implemented this method are as follows:

1. Non-intrusive trace collection requires a specialized debug hardware unit which our COTS platform do not provide.
2. Software-only trace solutions incur intolerable timing anomaly in execution traces.
3. RTBx is not an open source solution moreover it is a commercial product which is only usable with RapiTime to capture traces from the system.
4. Incurred probe effect with a mixed software/hardware solution is acceptable in our case.

Consequently, to be able to collect reliable measurements from the system, we needed to implement our own software/hardware measurement circuit design.

4.1 Minimizing Probe Effect for COTS Platforms

Betts et al. in [69] explain that software/hardware trace generation method is basically monitoring an outer interface of the platform and time stamping the trace identifiers by using an external hardware such as a Logic Analyzer. Instrumentation timing overhead is minimized since the only mechanism that is needed to be performed by the software is to output the trace identifier through one of the platform's output interfaces. He also states that this method requires accessible I/O ports on the target which are not always available and not practical to use. However, in our case the only acceptable solution is to use this software/hardware method along with the use of I/O ports which are available in our platform anyway.

There are several external interface options in order to output a trace identifier when a hit occurs. Those are UART, CAN, MIL-STD-1553, and GPIO. Trace identifiers are simply unique numbers and no other information is needed since time stamping is done externally hence the only thing that needs to be output are decimal trace identifiers. Because our main goal was to minimize the probe effect, we chose to implement a design which is based on GPIO interface. GPIO interface can be driven more easily rather than other options. Its timing penalty (overhead) and code size effect are much lower. Our LEON-3 based COTS platform has an 8 bits wide GPIO port in which each bit of the port can be individually set as input or output and can be controlled. An output register is used to drive the pin value so in order to output the trace identifier, the only thing that needs to be done in instrumentation code is writing a value to the GPIO output register. The limitation here is that we only have 8 GPIO pins which means our trace identifier can be represented with at most 8 bits.

When a signal appears at the external interface, a mechanism is needed to capture and time stamp the trace identifier. Mostly Logic Analyzers or Oscilloscopes are used in embedded systems to observe the signals and sometimes to capture them. However, in our situation both Logic Analyzers and Oscilloscopes do not have enough memory space to hold huge amount of trace data. Furthermore, these devices basically are not designed for that purpose. As a result, we needed to implement our own custom hardware to monitor the GPIO signals externally to capture, time stamp and store.

4.1.1 Design Overview

Instrumentation of the code side is easily done by just writing an 8-bit trace identifier value to the memory-mapped GPIO output register. However, designing custom monitoring hardware is much compelling. In our thesis work, we chose to implement our hardware with FPGA technology. That way, we could control and be aware of all the phases of capturing in order to increase reliability and minimize probe effect stemming from time stamping phase. Figure 4.1 shows the overall architecture of our trace capturing system. Target platform is connected to the TraceBox through its 8 GPIO pins. Our Host PC and TraceBox communicated through SSH over Ethernet interface so it is easy to remotely command the TraceBox and dump the logged traces. GDB Debug interface is used to load the instrumented program into the target platform and dump log data whenever necessary.

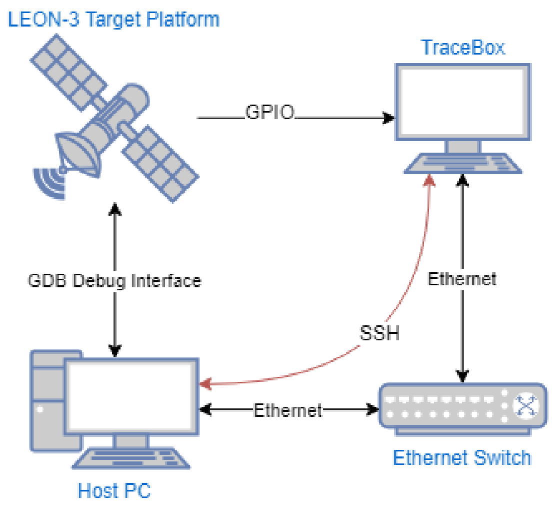


Figure 4.1: Overall Architecture of the Trace Capturing System

4.1.2 TraceBox Hardware Design

Monitoring GPIO signals and time stamping the changes can easily be done with any FPGA board. However, we also need to store the trace data to examine later after the tests are finished. Recently, an emerging technology is SoC FPGAs which integrates high-level management features of processors and real-time data processing capabilities of FPGAs into a powerful embedded computing platform [73].

In this work, we chose to implement our design on the Terasic DE0-Nano-SoC Kit which has Altera Cyclone V FPGA along with the 925MHz Dual-core ARM Cortex-A9 processor. It also has 1GB DDR3 SDRAM which we used to store our trace data temporarily until we move it to an external storage [4]. Figure 4.2 shows the block diagram of the kit.

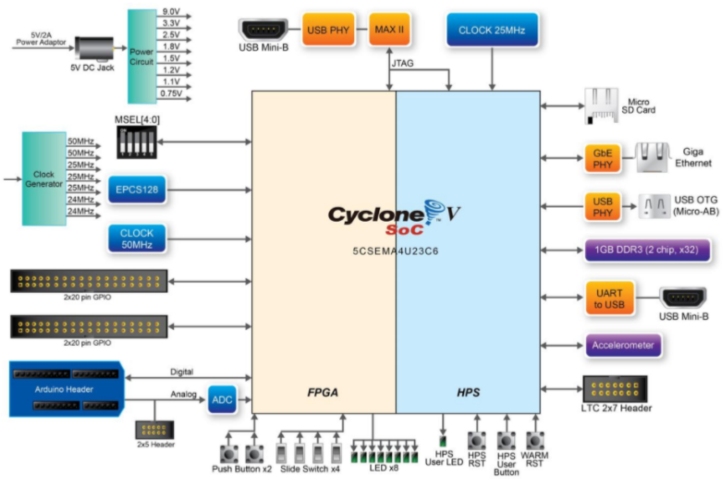


Figure 4.2: Block diagram of DE0-Nano-SoC [4]

Intel FPGAs have Avalon Interface capability which provides standard interfaces for designed components thus eases the development of FPGA components without worrying about their interconnections [74]. We designed our necessary components separately by conforming to Avalon Interface standards which is then connected easily with Platform Designer tool. Components and their interconnections are shown in Figure 4.3.

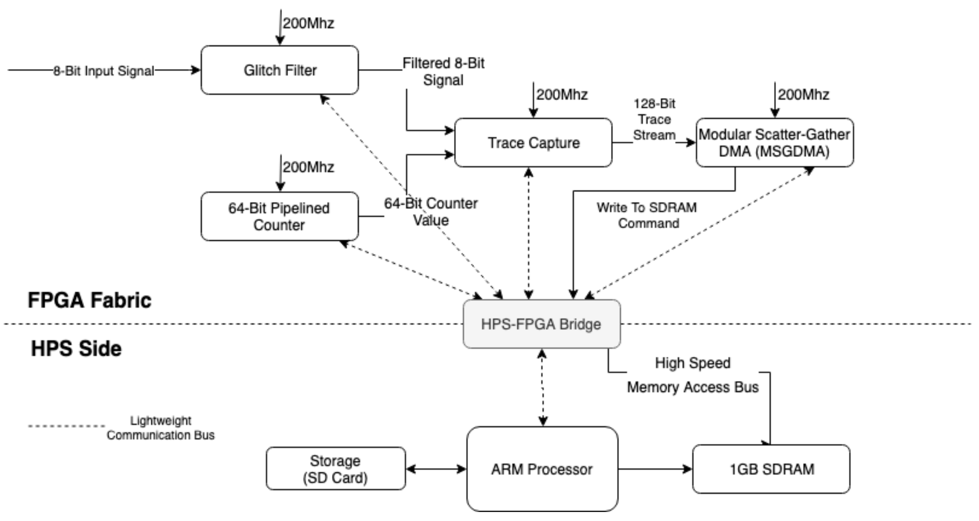


Figure 4.3: TraceBox Design Details

The main clock of the system is 50Mhz. However, by the help of PLL clock capability, we increased our system clock to 200Mhz as illustrated in Figure 4.3. All the components in the system are fed with this PLL clock. HPS-FPGA Bridge component provides the necessary mechanisms to access SDRAM at the HPS side of the board and it is also responsible for transporting commands from ARM processor to the FPGA fabric. MSGDMA component is basically a DMA which is wrapped with Avalon Interface standards and it is used to transport trace data directly to the SDRAM without visiting ARM itself thus minimizing the timing penalty resulting from storage phase. There are also some custom made components which are Glitch Filter, Pipelined Counter and Trace Capture.

4.1.2.1 Glitch Filter

Although our target platform's GPIO pins are equipped with two flip-flops in series to remove potential meta-stability, output signals might be exposed to environmental and electrical interferences. During our experiments, we observed too many false trace data if we splice the cables between the outputs of the onboard computer and the inputs of the TraceBox hardware. Most of the disturbances were vanished when we directly connected the pins through one whole wire per pin as shown in Figure 4.4.

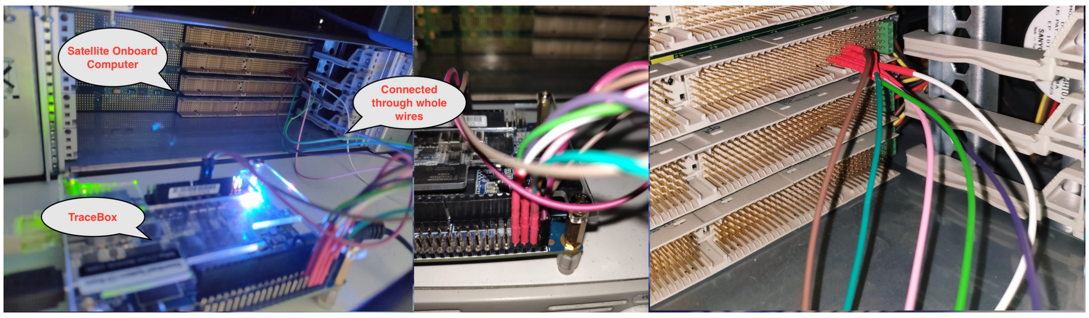


Figure 4.4: Connections Between the TraceBox and Satellite Onboard Computer

However, out of several hundreds of traces, there were still unwanted captures at the results. Glitch filter is designed to remove those unwanted pulse signals from the digital output signal. Its input signal width and glitch length can be configured through

ARM processor by the help of Avalon Standard thus relieving us from recompiling FPGA code for each possible configuration. Glitch length determines the number of samples that has to be stable before being propagated to the output. Figure 4.5 illustrates the filtering phase. As illustrated, received input signal must be stable for N clock cycles to be accepted where N is configurable through ARM processor.

The VHDL codes of Glitch Filter is given in Appendix B.1.

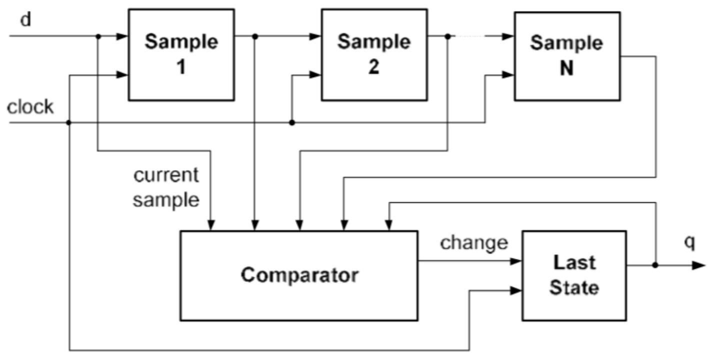


Figure 4.5: Block diagram of Glitch Filter

4.1.2.2 Pipelined High Speed Counter

FPGAs provide high speed counter capabilities. However, their reliability with higher speeds are questionable when conventional methods are being used such as defining a 64-bit signal vector and incrementing its value in each clock cycle. A 64-bit counter is essentially a 64-bit ripple carry adder which the *overflow* condition is compared in 64-bit at each cycle. For a 200Mhz clock source, this counting and comparing process should fit in 5ns. This would not be possible since the long counters have a lot of propagation delays. Thus, there needs to be a way to partition the counter signal into smaller parts and leading the use of faster FPGA counter blocks such as 16-bit counter blocks. Because of this reason we implemented a pipelined counter which the total length of the counter is divided into smaller blocks in which state transitions of blocks trigger subsequent blocks to start counting [75].

In our implementation, we chose to implement a 64-bit pipelined counter because 32-bit counter would roll over after 21.47 seconds with 200Mhz clock (4,294,967,295 x

5ns = 21.47s). In order to relieve us from tackling counter overflow issues, we chose to implement a 64-bit counter which would overflow after almost 3 years. We used 4 16-bit counter blocks in which rightmost 16-bit counter always counting at 200Mhz, but succeeding 16-bit counter increments only when the first 16-bit counter overflows and it continues until up to the leftmost 16-bit counter. In each clock cycle, 64-bit counter output is generated through Avalon interface which will be used by Trace Capture component.

The VHDL codes of Pipelined Counter is given in Appendix B.2.

4.1.2.3 Trace Capture

Trace Capture component involves the main capturing process of the TraceBox hardware. This component is basically waiting for an input signal transition to be captured. When the input is ready to be captured, it reads the 8-bit input value and current counter value from the Pipelined Counter component at the same time and finally sends the trace data to the MSGDMA component to be transferred to the SDRAM of the HPS. The destination address at the SDRAM is controlled from the HPS side of the system to avoid conflicts.

One important thing to consider in this scenario is how the Trace Capture component will decide whether the input signal is ready to be captured or not. There is not a communication protocol on top of the GPIO signaling mechanism. When a value is written to the GPIO pins, that value is latched until it is changed internally. Consumer side which is the TraceBox hardware needs to be informed that the data is written to the GPIO pins. For this reason, one of the GPIO pins is reserved to be the trigger pin which lowers the number of available pins for trace identification to 7.

In our implementation, most significant bit of the input signal is considered as trigger signal and rest of them are the trace identifiers. Figure 4.6 shows the timing diagram of the capturing and state transition phases.

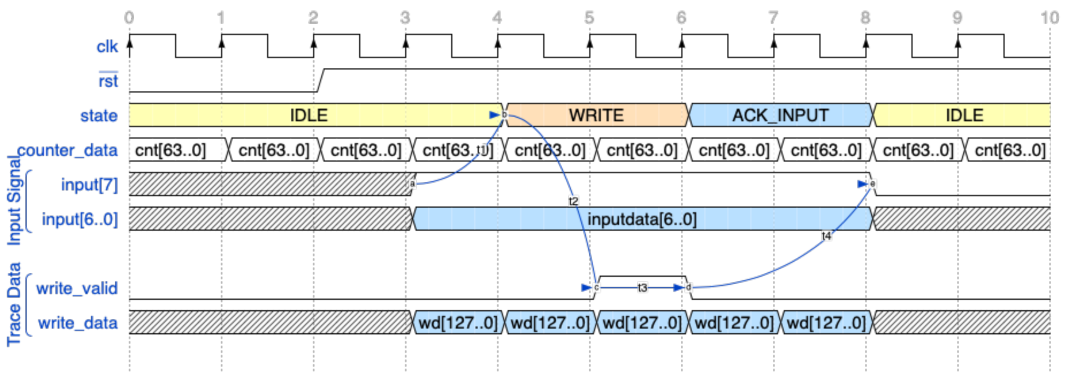


Figure 4.6: Trace Capture Timing Diagram

The VHDL codes of Trace Capture is given in Appendix B.3.

4.1.2.4 Software Instrumentation

As the most significant bit of the output signal needs to be used as trigger, available 7 pins could be used to identify instrumentation points over the program which makes it possible to inject up to 128 different points. Instrumentation points on software are defined as simple macros. The definition of the instrumentation macro is very simple that is just writing the identifier value to the GPIO output register. Listing 4.1b shows a simple instrumented program which basically calls IPoint macro two times when entering and leaving the program. Listing 4.1a shows the definition of **IPoint** macro which first writes the trace identifier value **x** to the output register. After that it toggles the most significant bit to high and eventually low to inform the external hardware that it finished writing trace identifier to the port.

Listing 4.1: Instrumentation Macro and Instrumented Program

(a) IPoint Macro

```
#define IPoint(x)
{
 GPIOOutput = (x & 0x7F);
 GPIOOutput = GPIOOutput | 0x80;
 GPIOOutput = 0;
}
```

(b) An Instrumented Program

```
void testProgram()
{
  IPoint(1);
  //program operations
  IPoint(2);
}
```

4.1.2.5 Storing and Dumping Traces

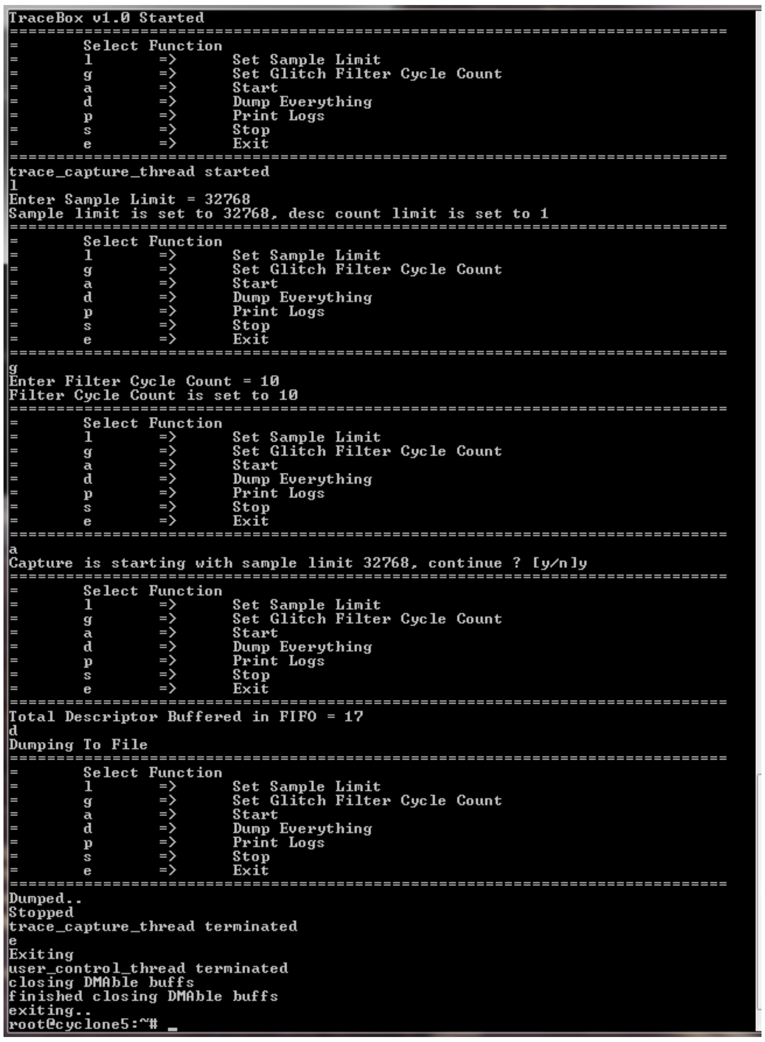
Trace Capture FPGA component captures the input signal as 8-bit signal and gets the current counter values as 64-bit data, but the SDRAM of the HPS side does not accept 64 + 8 = 72-bits of value for each entry. The nearest available option is to store the data as 128-bit data. For this reason, trace identifier is concatenated with 54 bits of zeroes to fill the 128-bit along with the counter value so each trace data occupies 128 bits (16 bytes) of space inside the SDRAM.

Our platform has 1GB SDRAM and theoretically it can hold 62,500,000 trace values. However, in practice this RAM is also used by the ARM processor on the board. Furthermore, even the 62,5 millions of samples seems enough, we do not want to limit ourselves with 1GB of space so we implemented a mechanism which fetches the trace data from SDRAM and writes to a file inside the memory card which is inserted to the Micro SD Card slot of the SoC board.

FPGA fabric and ARM processor works without disturbing each other in complete isolation. Hence, the software that is running on top of the ARM processor can work parallel with FPGA fabric. The only shared resource is the SDRAM between ARM and FPGA so it needs to be carefully crafted.

We allocated 16Mbytes of continuous ring buffer space from SDRAM since it is enough to store temporary trace values until they are stored inside the physical memory disk. A simple concurrent software running on top of the Linux operating system inside the ARM is responsible for configuring FPGA components through special Avalon interfaces and arranging MSGDMA destination addresses to avoid conflicts. In the mean time, it is also responsible for dumping trace values from SDRAM to the file.

A simple console interface is developed to control the software inside the ARM which is shown in Figure 4.7.



```
TraceBox v1.0 Started
============================================================================
=        Select Function
=        l        =>       Set Sample Limit
=        g        =>       Set Glitch Filter Cycle Count
=        a        =>       Start
=        d        =>       Dump Everything
=        p        =>       Print Logs
=        s        =>       Stop
=        e        =>       Exit
============================================================================
trace_capture_thread started
l
Enter Sample Limit = 32768
Sample limit is set to 32768, desc count limit is set to 1
============================================================================
=        Select Function
=        l        =>       Set Sample Limit
=        g        =>       Set Glitch Filter Cycle Count
=        a        =>       Start
=        d        =>       Dump Everything
=        p        =>       Print Logs
=        s        =>       Stop
=        e        =>       Exit
============================================================================
g
Enter Filter Cycle Count = 10
Filter Cycle Count is set to 10
============================================================================
=        Select Function
=        l        =>       Set Sample Limit
=        g        =>       Set Glitch Filter Cycle Count
=        a        =>       Start
=        d        =>       Dump Everything
=        p        =>       Print Logs
=        s        =>       Stop
=        e        =>       Exit
============================================================================
a
Capture is starting with sample limit 32768, continue ? [y/n]y
============================================================================
=        Select Function
=        l        =>       Set Sample Limit
=        g        =>       Set Glitch Filter Cycle Count
=        a        =>       Start
=        d        =>       Dump Everything
=        p        =>       Print Logs
=        s        =>       Stop
=        e        =>       Exit
============================================================================
Total Descriptor Buffered in FIFO = 17
d
Dumping To File
============================================================================
=        Select Function
=        l        =>       Set Sample Limit
=        g        =>       Set Glitch Filter Cycle Count
=        a        =>       Start
=        d        =>       Dump Everything
=        p        =>       Print Logs
=        s        =>       Stop
=        e        =>       Exit
============================================================================
Dumped..
Stopped
trace_capture_thread terminated
e
Exiting
user_control_thread terminated
closing DMAble buffs
finished closing DMAble buffs
exiting..
root@cyclone5:~# _
```

Figure 4.7: User Console Interface of TraceBox Software

User can set a sample limit, set the glitch filter length, start/stop capturing, dump remaining traces, print some debugging logs. It is accessible though SSH interface and resulting trace file can basically be copied to the host computer to be analyzed.

4.2 Evaluation and Comparison

In this section, we evaluated and compared our solution with widely used software-only instrumentation solution. Listing 4.2a shows our instrumentation macro, Listing 4.2b shows a widely used software-only instrumentation macro.

Listing 4.2: Instrumentation Macro Definitions of Compared Solutions

(a) Mixed Sw/Hw

```
#define IPointGPIO(x)
{
 GPIOOutput = (x & 0x7F);
 GPIOOutput = GPIOOutput | 0x80;
 GPIOOutput = 0;
}
```

(b) Sw-Only

```
buffer = allocate some space from
    the data memory space
#define IPointSW(x)
{
 currentTime = getTimeFromOS();
 write x to buffer;
 write currentTime to buffer;
}
```

Software-only solution requires some buffer space to be allocated from the memory and it makes a function call to the operating system to fetch the current time which is in general making a lot of inner function calls inside. After that trace identifier and current time value is written to the allocated buffer space until it is filled. Of course this software-only solution also requires a mechanism to dump this allocated buffer space to the host computer which our target platform has through GDB debug interface.

Out test methodology is to measure the difference of consecutive calls of the IPoint calls. What we have done is to prepare a test bench as shown in Listing 4.3.

Listing 4.3: Comparision Test Bench for Instrumentation Methods

```
IPointGPIO(1);
IPointGPIO(2);

IPointSW(1);
IPointSW(2);
```

As can be seen above, there are no other operations between IPoint calls and the only delay that incurred is their own execution times. IPointGPIO calls are captured through TraceBox hardware, IPointSW calls are stored inside the internal memory and then dumped though GDB debug interface when the test finishes.

We have collected 1000 measurements and the results are shown in Table 4.1.

Table 4.1: Precision Comparison of Trace Capture Methods

Methods	Min	Max	Avg
IPointSW	17 µs	50 µs	24 µs
IPointGPIO	0.86 µs	0.94 µs	0.89 µs

4.3 Conclusion

In order to collect reliable, minimal intrusive and higher number of traces, we have implemented an industrially viable solution for our thesis evaluation. Most of the currently used solutions store trace data in the memory to dump later. Although this method seems to be acceptable for end-to-end measurements, it is certainly not scalable and unacceptably intrusive to collect detailed traces, such as our case, where measurements of functional blocks inside the code is needed.

It is seen in Table 4.1 that our GPIO method outperformed the software-only method. It is much less intrusive that even in the worst case our method measured below 1 µs while the widely used software-only method took 50 µs to finish. Our mixed software-hardware solution is also able to capture limitless traces in theory as long as our storage is enough, which is replaceable (SD Card) while the on-board memory is not in most cases. Furthermore, the hardware is controllable remotely over SSH interface which we do not need to tackle the debug interface and stop the software and put the processor in debug mode in order to dump the traces stored internally. Hence, this method is used in our evaluation phase during all the experiments we carried out.

CHAPTER 5

HYBRID PROBABILISTIC TIMING ANALYSIS WITH EVT AND COPULAS

Hybrid Probabilistic Timing Analysis (HYPTA) has emerged to mitigate some of the current problems of measurement-only solutions by combining static properties with measurements. Static properties expose the structure which makes it possible to analyze parts of the program individually and possibly increase path coverage by analytically constructing new paths.

In Static Probabilistic Timing Analysis (SPTA), all the instructions are attached with an Execution Time Profile (ETP) which the execution times and related probabilities are assigned under the probabilistic assumptions based on instruction type. For example, if an instruction is data dependent, meaning that if it should access to the memory, its execution time could be hypothetically 1 or 100 cycle when the corresponding data is already in cache or not. The probabilities of that instruction to execute in 1 or 100 cycles are assigned respectively based on historic assumptions that have been made before. In any case, probabilities and execution time values inside the ETPs are constructed using all static and some probabilistic assets.

HYPTA approach also divides the program into predefined blocks. However, ETPs are constructed from the measurement observations. After constructing ETPs, an approach is needed to combine those ETPs to derive an execution time distribution of the whole program that is being analyzed.

In this chapter, we propose our Hybrid Probabilistic Timing Analysis framework. Instead of representing each block with their ETPs, we propose to represent them with continuous parametric probability distribution functions and specifically with

Extreme Value Distribution functions whenever possible. Since this hybrid approach is an improvement over a state-of-the-art hybrid probabilistic method, we also provide mechanisms to use ETPs when the blocks are not suitable to be modeled by an EV distribution. Furthermore, we provide the details of the application procedure of Copulas in timing analysis to model the dependency between the blocks.

5.1 Current State-of-the-Art

There are very few studies in HYPTA domain to mention. The most conspicuous works are done by $i)$ Kosmidis et al. [53] called as *Path Upper Bounding (PUB)*, which is referred as the main HYPTA technique by Abella et al. [76] and $ii)$ *Extended Path Coverage for MBPTA (EPC)* [54]. They both aim at increasing the path coverage for MBPTA by using static properties of the programs. PUB simply relies on modifying the source code which requires specialized compiler support to force the program to traverse uncovered paths during the analysis runs while the EPC synthetically pads the execution time distributions of each block in order to make each block independent of each other and finally combines them to derive the worst-case path for the program.

However, our main interest is on the RapiTime tool which is developed by Rapita Systems Ltd. [49]. RapiTime is an industrial probabilistic WCET solution which its roots reach originally to the pWCET Tool [30]. It was initially a research project developed by Real-Time Systems Research Group at the University of York [46]. Throughout several research studies [10, 47, 30, 9, 77], the tool has evolved into a commercial product now called as RapiTime.

Authors of [2] defines the RapiTime as a hybrid MBTA solution rather than hybrid MBPTA. They argue that the tool predicates the notion of pWCET, but it does not apply a predictive model to estimate the distribution. Thus, it should be considered much more as a SPTA with measurements rather than MBPTA. However, Davis et al. [46] states that the tool falls into the HYPTA category since it combines the static structural properties of the program with measurements, but its probabilistic approach is still questionable. Besides, one of the developers of the tool states that they follow a frequentist approach in order to determine the probability values of each measurement sample [30].

Cazorla et al. [2] describes the ETP for an instruction x as:

$$ETP(x) = \begin{pmatrix} p_{x_1} & p_{x_2} & .. & p_{x_n} \\ et_{x_1} & et_{x_2} & .. & et_{x_n} \end{pmatrix}, \sum_{i=1}^{n} p_{x_i} = 1.$$

where et_{x_i} corresponds to a possible latency of the instruction and p_{x_i} represents its probability of occurrence. The RapiTime tool constructs this ETP for each block but not for every instruction. et_{x_i} values represent the observed unique execution times and p_{x_i} is their observation frequency.

A novel approach to combine ETPs in HYPTA domain is described in [30] and we use the term HYPTA to represent their approach in our work. They state that any type of program can be described as a combination of sequential, conditional and iterative blocks. Their representation method is named as Scope-Tree and described in [47]. An important criteria in HYPTA different from SPTA is that ETPs do not represent the execution-time behavior of the individual instructions, but rather basic blocks or coarse grained functional blocks. One of the main reasons is that capturing measurements for each instruction is not a trivial task and also there would not be enough variability in the execution time of all individual instructions during analysis runs.

They defined a set of rules called as the *timing schema* of the program in order to evaluate the WCET as a function of tree nodes [9]. Tree nodes correspond to measurable blocks inside the program and each node should have an ETP that will eventually be combined to construct the overall WCET. The notions that are used to represent a basic program are given as follows:

- $W(A) = X$
- $W(A; B) = W(A) + W(B)$
- $W(\text{if E then A else B}) = W(E) + max(W(A), W(B))$

- $W(\text{for E loop A end loop}) = W(E) + n(W(E) + W(A))$, where n is the maximum iteration count

X is a random variable representing the execution time of node A. $W(A; B)$ corresponds to sequential execution of nodes A and B. The third notion represents a conditional statement and the last one is used for sequential blocks. They state that this scheme (conditional, sequential, iteration) is enough to represent any type of structured program [9].

Here, the resulting expression will be the probabilistic timing schema of the worst-case path of the whole program. An important point in here is that the method actually derives possible worst-case paths and their individual WCETs construct the overall distribution for the program. However, numerical calculation of the expression is not an easy task especially for sequential cases.

Since the execution time of the nodes are represented by random variables X_i, the problem reduces to the calculation of $(X_1+X_2+..+X_n)$ for sequential blocks. If it is assumed that X_is are independent of each other, then the standard convolution of the distributions gives the result easily. However, this assumption do not hold in reality especially for the perfect positively(comonotonic) or negatively(countercomonotonic) dependent situations. Petters et al. [78] compares several ETP convolution techniques and he also remarks the issue of the overestimation of current approaches.

Bernat et al. [9] in their study specifically aim to solve this problem by using copulas. Copulas are basically used to model the dependency between random variables [79]. The study proposes that the supremal convolution with the assumption of comonotonicity between the blocks results in safe estimation for any type of dependence between them. It means that whether the sequential blocks are independent or not, comonotonic convolution safely upper bounds the results of any possible degree of dependence and they experimentally confirmed the idea. Similar to the supremal convolution, they branded the Biased Convolution technique [10] which still relies on comonotonicity. Furthermore, their commercial tool RapiTime also implements the explained Biased Convolution method [49].

5.2 Open Challenges for HYPTA

There are some problems and open challenges for the current state-of-the-art HYPTA approach. First, in one of their constituent papers [9], they state that the only acceptable method is comonotonic convolution for any type of dependence between basic blocks. They support the idea with some experiments, but the degree of overestimation is not mentioned. Actually the results in the paper do not seem to be overestimated because of the test scenarios which only covers the comonotonic and independent cases. However, for a countercomonotonic case (perfect negative dependence), the assumption of comonotonicity would result in huge overestimation [10]. Consider the following program:

Listing 5.1: Negatively Dependent Sequential Function Call Example

```
void f1(x) {
  for(int i = 0; i < x; i++)
  {
    //delay 1ms
  }
}

void f2(x) {
  for(int i = 0; i < 100 - x; i++)
  {
    //delay 1ms
  }
}

void testProgram()
{
  f1(x);
  f2(x);
}
```

It is obvious that the functions $f1$ and $f2$ are negatively dependent to each other meaning that when one of them executes for (n) ms , the other one should execute for $(100 - n)$ ms which results in $testProgram$ always executing for $100\,\mathrm{ms}$. Current HYPTA solution tries to estimate the WCET of the $testProgram$ function by calculating the following expression.

$$W(testProgram) = W(f1) + W(f2)$$

Here, $W(f1)$ and $W(f2)$ are the observed execution time distributions that are represented by random variables. Sequential addition of the random variables with comonotomic convolution results in:

$$W(testProgram) = ETP\{testProgram\} = \begin{pmatrix} p_1 & p_2 & .. & p_n \\ 2 & 3 & .. & 200 \end{pmatrix}$$

The resulting distribution claims that the WCET of $testProgram$ could be $200\,\mathrm{ms}$ with a probability of p_n or alternatively it claims that there could be an input x which would lead the program through a worst-case path where both $f1$ and $f2$ executes for $100\,\mathrm{ms}$. This assumption is far from reality and causes to 100% overestimation of the actual WCET.

The second problem is related to the construction of ETPs for each basic block. The method is based on a frequentist approach meaning that the frequency of observation of each value is assigned as the probability of occurrence for each value. For example, lets consider the execution time observations for block A corresponds to:

$$X_A = \{10, 10, 11, 11, 11, 11, 12, 13, 14, 14\}$$

The ETP for the block A is constructed as:

$$ETP(A) = \begin{pmatrix} 0.2 & 0.4 & 0.1 & 0.1 & 0.2 \\ 10 & 11 & 12 & 13 & 14 \end{pmatrix}$$

where the values in the first row corresponds to the probabilities and the second row represents the possible execution times for the block. It is also stated that the distributions for blocks are discrete in nature. This statement holds for fine grained basic blocks since they are not exposed to huge variations. However, if the blocks are defined as functions, they are much vulnerable to input and hardware effects, thus their execution time behavior would result in an asymptotic tail as was examined in chapter 3.

In practice, it is not feasible to instrument every branching points through the program in order to construct ETPs for fine grained basic blocks due to increasing probe effects as was explained in chapter 4. Consider the following program with nested loops:

Listing 5.2: Function with Nested Loops

```
void f1() {
  for(int i = 0; i < 100; i++)
  {
    IPoint(1)
    for(int j = 0, j < 100; j++)
    {
      IPoint(2)
      ...
      IPoint(3)
    }
    IPoint(4)
  }
}
```

For only one call of function $f1()$, instrumentation methods $IPoint(2)$ and $IPoint(3)$ are called for 10000 times which exposes the infeasibility of the approach especially for COTS platforms. The only reasonable solution would be to increase the granularity of the blocks which also brings the variance issue due to multi-path nature of coarse grained blocks.

RapiTime allows to instrument the program in functional granularity which means that only the entry and exit points of the functions are instrumented throughout the program. However, they still construct the ETPs in the same way as basic blocks. This approach ignores the possible rare events due to hardware and input effects of the functions which generally have a multi-path nature.

The aim of this present work results from the open challenges of current state-of-the-art HYPTA solution which eventually turned into a commercial probabilistic WCET tool named as RapiTime. Those problems are not raised from the nature of the methods that are being implemented, but the applied domain and system wide constraints such as hardware/software limitations that result in the lack of applicability and reliability of the obtained results.

The overestimation resulting from the assumption of comonotonicity and the lack of rare event capturing when functional level instrumentation is performed has emerged as the main research topic of this study.

5.3 Estimating Tighter pWCET

The main challenge in measurement based WCET analysis era is to derive reliable and tighter WCET values or distributions. As the actual WCET value is not known in practice, there is no known method to validate the estimated value or its tightness. However, in some cases it is instinctively known that the implemented method results in overestimation especially when the selected method chooses to stay in the safe side ignoring the incurred pessimism.

In our case, we aim to decrease the unnecessary overestimation that result from the comonotonic assumption of the current state-of-the-art HYPTA solution. It is not our claim that the existing method could result with an overestimation, in fact its developers also admit it [9, 10]. The main idea behind assuming comonotonicity arises from the fact that generally the dependence information between the blocks are not known so the comonotonic convolution is the only safe estimate for finding the joint distribution of basic blocks.

Consequently, when the dependence between the blocks are known, comonotonic assumption could be eliminated. Deriving a dependence information for the distributions is not a trivial task, but possible. If the execution time of the basic blocks are considered as random variables X_i then the problem reduces to finding a dependence type between consecutive random variables. Bernat et al. [9] study the issue by using Copulas, but only examine some specific dependence types. In reality, copulas can model a wide range of dependencies. Details of the Copulas were already given in chapter 2.

It is also possible to model each X_i with a parametric distribution such as GEV or GPD in order to capture possible extreme cases at the tails. Modelling the distributions were detailed in chapter 3 and the general steps are also valid in this case.

Recently, combining Extreme Value Theory and Copulas became an emerging subject especially in economics and finance [11, 79]. In our work we followed a similar approach with [11].

5.3.1 Proposed Method

In finance, Value-at-Risk (VaR) is an important measure that represents the risk factor of the portfolio. Basically, VaR is a single number that gives the amount of potential loss that could happen in an investment or a portfolio of investments over a given time period. Formally,

$$VaR_\alpha = inf\{l \in \mathbb{R} : P(L > l) \leqslant 1 - \alpha\} = inf\{l \in \mathbb{R} : F_L(l) \geqslant \alpha\}, \alpha \in (0, 1)$$

gives the VaR of a portfolio at the confidence level α, which is the smallest l such that the probability of the loss L exceeding l is no larger than $(1 - \alpha)$. Figure 5.1 illustrates the possible loss distribution for a given time period and red dashed line represents the VaR value with $\alpha = 95\%$ meaning that the loss value will not be greater than VaR with 95% probability. Alternatively the loss to be grater than VaR by 5% chance.

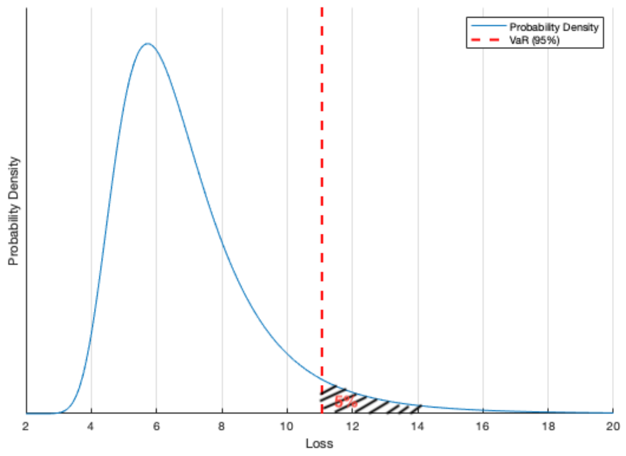


Figure 5.1: Value at Risk

Deriving the VaR value requires to construct a loss distribution. If the aim was to estimate the VaR value for only one investment, the problem reduces to model the loss/return indexes of the investment and calculate the 95% quantile of the distribution. However, modelling the distribution for a portfolio of investments is a challenging task due to the dependency between investment returns.

In order to model the distribution, there are several methods; namely Historical Simulation, Variance-Covariance Method and Monte-Carlo Simulation. In our work, we take advantage of Monte-Carlo Simulation approach because it provides probabilistic results and the interdependence between individuals are taken into account.

There are also other algorithms to numerically calculate the joint distribution of n dependent random variables in the literature such as the GAEP algorithm by Arbenz, Embrechts and Pucetti in [80]. However, their method is limited to calculate the joint distribution of 6 marginals and the results seem to be very similar to Monte-Carlo method.

Avdulaj in his MSc thesis [11] proposes a procedural approach to estimate the VaR for an empirical portfolio by using Extreme Value Theory and Copulas. The general steps that he followed which also concern us are given as follows:

1. Model each investment returns with Piecewise Distribution with Pareto Tails which fits a Generalized Pareto Distribution (GPD) to the tails and a kernel distribution to the intermediate part.

2. By using semi-parametric CDF that is derived in the first step, each distribution is transformed into a uniform one on interval $[0, 1]$.

3. A multivariate t-copula is fit to uniform marginal distributions and its parameter is estimated.

4. By the help of the t-copula generator function, huge number of uniform variates are generated randomly.

5. Randomly generated uniform variates are converted back to their real domain by using the inverse of each semi-parametric distribution which is also called as Inverse Transform Sampling in the literature [81].

6. A weighted sum is performed to estimate the overall VaR distribution.

7. Finally, the desired VaR value is calculated with the given confidence level α.

The steps after the $3rd$ one represents the Monte-Carlo Simulation steps.

An analogy can easily be drawn between VaR analysis and the WCET analysis in our case. Returns of investments correspond to execution time of blocks and VaR corresponds to the overall probabilistic WCET of the program.

In our proposed method, we follow a similar approach as Avdulaj used in his MSc thesis, but instead of using t-copulas to model the dependence, we chose to use Vine Copulas. Although the t-copula allows to model symmetric tail dependencies in higher dimensions, it still relies on a single parameter. In fact, multivariate dependencies are not necessarily symmetric. Also when the dimensions become more complex, single parameter approach might fall behind. For these reasons, Vine Copula approach is introduced which models the overall dependency by using pair-copulas and a tree model [82, 83].

Additionally, we also model each block of the program with Extreme Value distributions. However, there might be some cases when goodness-of-fit tests are not passed for the fitted models as was explained in chapter 3. In such cases historical simulation approach is followed which corresponds to using frequentist ETPs that are introduced before. Modelling the blocks with one of the EVT distributions provides deriving predictive tail values while current state-of-the-art ETP approach only provides already observed discrete values.

The most essential step in our method is to derive the copula model which represents the dependency between scopes of the analyzed program. In order to derive a copula, marginals must have the same dimensions meaning that the observations of each scope must be taken at the same time. This is an important criteria especially for conditional and iterative blocks since for a single run of the program, sequential blocks are visited once, but conditional or iterative blocks might be visited zero or several times. The main aim of this study is to reason about the dependencies for sequential blocks and derive tighter bounds for the WCET, thus a special consideration is taken for the conditional and iterative blocks.

The general steps of our approach is given below.

1. Derive the timing schema of the program and identify the scopes to be analyzed

separately

2. Determine the random variables that represents the execution time of the scopes
3. From the most inner scope to the outer one, fit the suitable copulas for each scope that have sequential addition of random variables
4. Simulate next n execution of the scopes out of the copulas by using Monte-Carlo approach
5. Derive the inverse CDFs of each random variable
6. Transform each uniform margins generated from simulations into their original domain by using their inverse CDFs
7. Perform a sum operation for all the marginals to derive the overall distribution of the scope
8. Repeat steps $2 - 7$ until there are not any scopes left to be analyzed

5.3.2 Experimental Evaluation

The purpose of this section is to identify the experimental setup and detail the application steps of our method. In addition, we compare our results with the current state-of-the-art solutions.

In this chapter we use the same setup as explained in chapter 3 - algorithm 1. However, in this case traces are not stored internally, but they are captured by using our custom developed TraceBox device as detailed in chapter 4. Using such a solution is mandatory since it is necessary to obtain detailed traces instead of end-to-end measurements as in chapter 3. However, as was noted earlier instrumenting every basic block or every branch point is not feasible due to the possibly incurred probe effect and hardware limitations (i.e. limited GPIO pins). Therefore, we chose to instrument the programs in functional block granularity.

Captured traces are analyzed on our host computer after the analysis runs are finished. MATLAB and R languages are used to analyze the traces and visualize the results.

Most of the steps in this chapter are done in R environment since it supports a wide range of statistical algorithms and aesthetic data visualization capabilities [84].

5.3.2.1 Application Procedure

In order to detail the steps, we constructed a program similar to the program given in Listing 5.1. In that case we already mentioned about the huge overestimation problem. However, for more realistic scenarios, it would not be easier to reason about the overestimation cause. In fact, we cannot even be sure about whether there is an overestimation or not. The only implication could be drawn from a detailed code analysis of the program. Consider the following program:

Listing 5.3: Sequential Negatively Dependent Insertion Sort

```
void testProgram()
{
  insertSortAsc(x);
  insertSortDesc(x);
}
```

The given program is composed of two consecutive sorting functions both depending on the given input x. x is a randomly generated array that is composed of 500 float values. The first function sorts the array in ascending order and the second one sorts the same array in descending order. Since x is randomly generated for each run, their execution times would be negatively correlated. In addition to that those functions are also input dependent multi-path programs which have loops and floating point comparisons. For reproducibility purposes, the sorting function is realized by the *insertsort* program out of the Mälardalen WCET Benchmarks [62].

insertsort is originally an insertion sort program which sorts a reversed array of size 10. It is an input-dependent program with nested loops. For our evaluation we extended the sort bound from 10 to 500 and for descending ordering case we only changed the comparison condition from "<" to ">".

Instead of instrumenting only the *testProgram*, this time we instrumented the program in functional granularity which means the *insertSortAsc* and *insertSortDesc* functions are also instrumented along with the *testProgram* itself. Out of 10000 analysis runs, observed statistics is given below.

Table 5.1: Execution Time Statistics of the Observations

Scope	*Minimum*	*Maximum*
insertSortAsc	217.4657 ms	272.0894 ms
insertSortDesc	212.7195 ms	266.9449 ms
testProgram	482.2895 ms	487.3415 ms

It is clearly seen from Table 5.1 that although both sorting functions execute for more than 260 ms, the observed maximum end-to-end execution time for testProgram is about 487 ms. This indicates that the functions are not perfectly positively correlated (comonotonic), thus the dependency type between them should be taken into account. The correlation plot given in Figure 5.2 clearly summarizes the case.

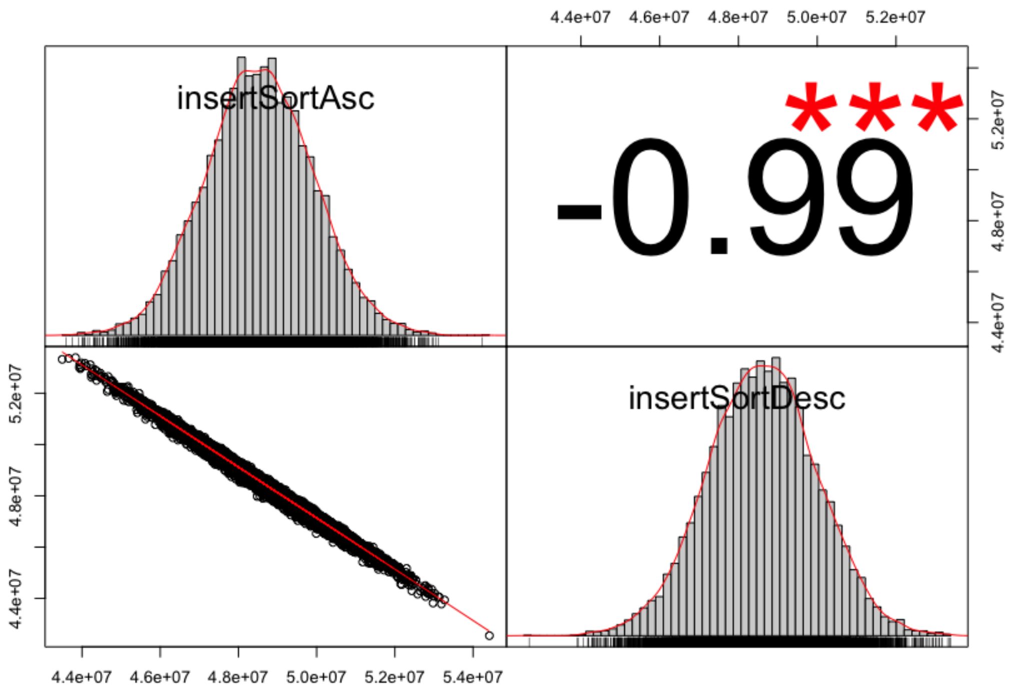


Figure 5.2: Correlation Plot of Negatively Dependent Sorting Functions

The distribution of each function in histogram format and kernel density estimation line is shown on the diagonal. On the bottom of the diagonal, the bivariate scatter plots with a fitted line is displayed. On top of the diagonal, the value of the correlation coefficient and the significance level as stars is shown. The significance level of the relationship is associated to a symbol: p-values$(0, 0.001, 0.01, 0.05, 0.1, 1)$ <=> $("***", "**", "*", ".", "")$. Correlations with p-value > 0.01 is considered as insignificant so the symbols are left blank. Note that the values on both axes correspond to clock tick counts in which each tick represents $5\,\mathrm{ns}$ that is the resolution of our TraceBox unit.

The plot in Figure 5.2 illustrates the correlation between the functions. However, our *testProgram* function also has a self execution time which is the execution time when the execution time of individual functions are excluded from end-to-end measurements. Since our *testProgram* is composed of only 2 function calls, that self execution time is negligible, but we still take that into account.

5.3.2.1.1 Timing Schema of the Program

Next step is to derive the timing schema of our program. The instrumented version of the program is given below :

Listing 5.4: Instrumented Negatively Dependent Insertion Sort

```
void insertSortAsc(x)
{
  IPoint(29);
  //sort x in ascending order
  IPoint(28);
}

void insertSortDesc(x)
{
  IPoint(27);
  //sort x in descending order
  IPoint(26);
}

void testProgram()
{
  IPoint(31);
  insertSortAsc(x);
  insertSortDesc(x);
  IPoint(30);
}
```

The timing schema in the form of random variables and their corresponding IPoint pairs are shown in the following expressions:

$$W(insertSortAsc) = W(IPoint_{29-28}) \quad (51)$$

$$W(insertSortDesc) = W(IPoint_{27-26}) \quad (52)$$

$$W(testProgram_{entry}) = W(IPoint_{31-29}) \quad (53)$$

$$W(insertSortAsc_{ret}) = W(IPoint_{28-27}) \quad (54)$$

$$W(insertSortDesc_{ret}) = W(IPoint_{26-30}) \quad (55)$$

where

- $W(testProgram_{entry})$ = Time passed until $insertSortAsc$ starts executing from the start of the $testProgram$
- $W(insertSortAsc_{ret})$ = Time passed until $insertSortDesc$ starts executing from the return of the $insertSortAsc$
- $W(insertSortDesc_{ret})$ = Time passed until $testProgram$ returns from the return of the $insertSortDesc$

Both $W(insertSortAsc)$ and $W(insertSortDesc)$ are equal to the end-to-end execution time of the functions since they do not contain sub function calls inside. The total execution time of the $testProgram$ can be represented as:

$$W(testProgram) = \underbrace{W(testProgram_{self})}_{\text{X}} + \underbrace{W(testProgram_{sub})}_{\text{Y}} \quad (56)$$

where $W(testProgram_{sub})$ corresponds to the timing schema of the sub-function calls inside the $testProgram$ and $W(testProgram_{self})$ represents the timing schema of the main body of the program when sub-functions are excluded. Thus, they can be expressed as:

$$W(testProgram_{self}) = \underbrace{W(testProgram_{entry})}_{\text{X}} + \underbrace{W(insertSortAsc_{ret})}_{\text{Y}} + \underbrace{W(insertSortDesc_{ret})}_{\text{Z}} \quad (57)$$

$$W(testProgram_{sub}) = \underbrace{W(insertSortAsc)}_{\text{X}} + \underbrace{W(insertSortDesc)}_{\text{Y}} \quad (58)$$

5.3.2.1.2 Deriving The Copulas

Each equation 56, 57 and 58 are composed of sub-parts which are represented by random variables X, Y and Z which might be the nodes or leaves of the program structure tree. Each random variable represents the execution time of an inner scope of their parent scope. For example, in equation 56 the program is divided into 2 sub-scopes. Thus, from the scope of $testProgram$, there are only 2 parts which are represented by X and Y, respectively. Thus, the problem reduces into the sum of two random variables where the dependency between them is unknown. Identically, in equation 57 there are 3 sub-scopes, and in equation 58 there are 2 sub-scopes. Each equation is a problem of sum of n random variables and the joint distribution H is needed.

Considering equation 57 which have the highest number of variables, in order to derive the multivariate joint distribution $H(x, y, z) = P[X \leq x, Y \leq y, Z \leq z]$ which is necessary to finally derive the $J(t) = P[X + Y + Z \leq t]$, copulas are suitable since according to Sklar's theorem for any $(u, v, k) \sim U(0, 1)$:

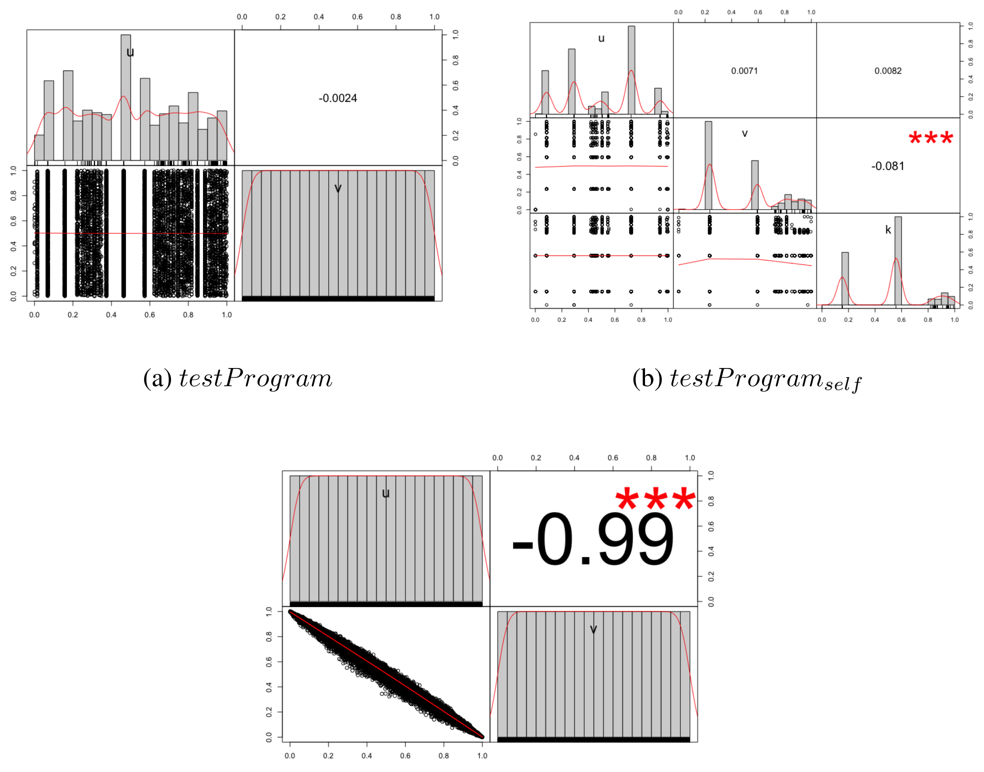
$$H(x, y, z) = C(F(x), G(y), M(z)) \quad (59)$$

$$C(u, v, k) = H(F^{(-1)}(u), G^{(-1)}(v), M^{(-1)}(k)) \quad (510)$$

where F, G and M corresponds to the cumulative distribution functions of X, Y and Z, respectively. Additionally u, v and k should have the same dimensions which in our case are the execution time observations for the corresponding scopes.

As in line with chapter 3 - algorithm 1, *testProgram* is run for 10000 times. When $testProgram$ is run for 10000 times, same amount of observations are collected for both $W(testProgram_{self})$ and $W(testProgram_{sub})$. Similarly, 10000 samples are taken for both $W(insertSortAsc)$ and $W(insertSortDesc)$ when $W(testProgram_{sub})$ is visited 10000 times. Same logic applies to the sub-scopes of the $W(testProgram_{self})$.

Vine Copula package of R is used to fit a copula to the marginal distributions which in this case are X, Y and Z. To do this, first the marginals should be converted into uniform range. Original observations are converted into uniform distribution by using `pobs` function which computes the pseudo-observations for the given data matrix in Vine Copula package. After the conversion, the correlation between the sub-scopes of each scope becomes:



(a) $testProgram$

(b) $testProgram_{self}$

(c) $testProgram_{sub}$

Figure 5.3: Correlation Plot of the Pseudo-Observations

Note that the dependency between $u(pobs_{insertSortAsc})$ and $v(pobs_{insertSortDesc})$ in Figure 5.3c coheres with Figure 5.2. However, the dependency between the sub-scopes of $testProgram$ in Figure 5.3a seems to be independent because there are no operations between the functions which might result in any type of behavior. This situation also results in almost independence between the sub-scopes of $testProgram_{self}$ that is seen in Figure 5.3b.

Next, `RVineStructureSelect` function of Vine Copula package is used to fit a suitable copula model to our program scopes. It also selects the tree structures that are appropriate for the pair-copula families. The output of the function in R environment is given below.

Listing 5.5: RVineStructureSelect Results of Each Scope

```
> RVM_testProgram
C-vine copula with the following pair-copulas:
Tree 1:
1,2  Independence
---
1 <-> u,   2 <-> v

> RVM_testProgramSelf
C-vine copula with the following pair-copulas:
Tree 1:
3,1  Independence
3,2  Gaussian (par = -0.13, tau = -0.09)
Tree 2:
2,1;3  Independence
---
1 <-> u,   2 <-> v,   3 <-> k

>  RVM_testProgramSub
C-vine copula with the following pair-copulas:
Tree 1:
1,2  t (par = -0.99, par2 = 30, tau = -0.94)
---
1 <-> u,   2 <-> v
```

It can be seen from the output that an independence copula is fitted to the sub-scopes of $testProgram$ and $testProgram_{self}$ as expected. Gaussian copula with parameter -0.13 corresponds to the almost independence case. Important result is that a t-copula with parameter $\{-0.99\}$ is fitted to the (u, v) pairs of $testProgram_{sub}$ which also coheres with the results given in Figure 5.3c.

Next step is to check whether this copula structure is suitable for the given data sets. In order to do that the `RVineGofTest` function is used which performs 15 different goodness-of-fit tests for R-Vine copula models. In our evaluation *ECP2* method is chosen which is a goodness-of-fit test based on the combination of probability integral transform (*PIT*) and empirical copula process (*ECP*) [85]. The output of the goodness-of-fit for the R-Vine copula is given below.

Listing 5.6: RVineGofTest Results for the derived Vine Copulas

```
> RVineGofTest(df_testProgram, RVM_testProgram, method="ECP2", statistic =
    "CvM")
$CvM
[1] 3.869722
$p.value
[1] 0.57

> RVineGofTest(df_testProgramSelf, RVM_testProgramSelf, method="ECP2",
    statistic = "CvM")
$CvM
[1] 236.164
$p.value
[1] 0.445

> RVineGofTest(df_testProgramSub, RVM_testProgramSub, method="ECP2",
    statistic = "CvM")
$CvM
[1] 6.837797
$p.value
[1] 0.585
```

The p-value which is greater than 0.05 indicates that there is no significant evidence against rejecting the fitted copula that is suitable to model the dependency between the sub-scopes of the given scopes. Results show that all copulas that are fitted to $testProgram$, $testProgram_{self}$ and $testProgram_{sub}$ are valid.

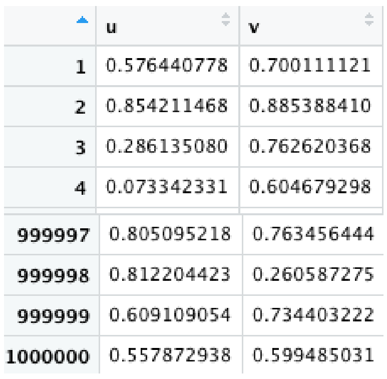
5.3.2.1.3 Simulation From The Copulas

By using these copula models, the Monte-Carlo simulation steps are followed basically by randomly generating uniform probability values. To do that, `RVineSim` function is used which generates desired number of probability value pairs out of the given R-Vine copula model. In our experiments we generated $1.000.000$ samples which actually represents the next one million possible outcomes of the program scopes in accordance with the dependency model.

Listing 5.7: Simulation from the R-Vine Copula

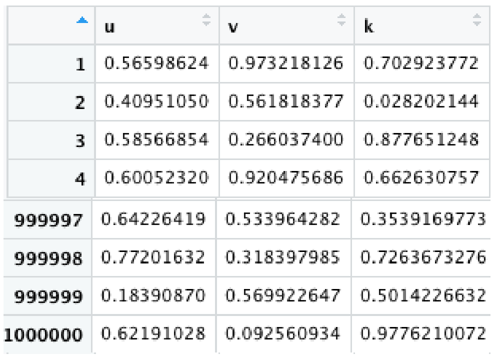
```
> simdata_testProgram <- RVineSim(1000000, RVM_testProgram)
> simdata_testProgramSelf <- RVineSim(1000000, RVM_testProgramSelf)
> simdata_testProgramSub <- RVineSim(1000000, RVM_testProgramSub)
```

The results of the simulations are $1.000.000$ x n matrices where n corresponds to the number of marginals of each copula. Simulated uniforms can be illustrated as:



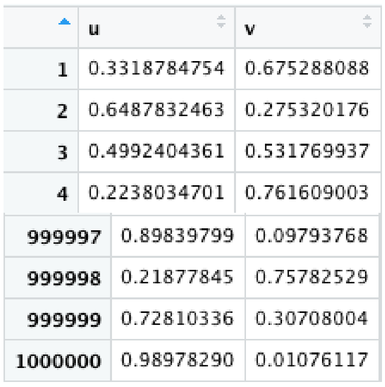
	u	v
1	0.576440778	0.700111121
2	0.854211468	0.885388410
3	0.286135080	0.762620368
4	0.073342331	0.604679298
999997	0.805095218	0.763456444
999998	0.812204423	0.260587275
999999	0.609109054	0.734403222
1000000	0.557872938	0.599485031

(a) $testProgram$



	u	v	k
1	0.56598624	0.973218126	0.702923772
2	0.40951050	0.561818377	0.028202144
3	0.58566854	0.266037400	0.877651248
4	0.60052320	0.920475686	0.662630757
999997	0.64226419	0.533964282	0.3539169773
999998	0.77201632	0.318397985	0.7263673276
999999	0.18390870	0.569922647	0.5014226632
1000000	0.62191028	0.092560934	0.9776210072

(b) $testProgram_{self}$



	u	v
1	0.3318784754	0.675288088
2	0.6487832463	0.275320176
3	0.4992404361	0.531769937
4	0.2238034701	0.761609003
999997	0.89839799	0.09793768
999998	0.21877845	0.75782529
999999	0.72810336	0.30708004
1000000	0.98978290	0.01076117

(c) $testProgram_{sub}$

Figure 5.4: Simulated Uniform Margins from the Fitted Copula Models

It can be observed from Figure 5.4c that the values of u and v follow a negatively correlated nature while the pairs in Figure 5.4a and 5.4b behaves independently as expected. Those simulated uniform values represent possible outcome of the next execution of the corresponding scope. For example, the first line in Figure 5.4c states that the next execution of $testProgram_{sub}$ would result in execution time of $insertSortAsc$ to be $F^{(-1)}(0.3318784754)$ and the execution time of $insertSortDesc$ to be $G^{(-1)}(0.675288088)$ where F and G corresponds to the CDFs of the scopes. Thus, it is important to derive the CDFs of each scope in order to construct a safe upper bound for the $testProgram$.

5.3.2.1.4 Deriving CDFs of the Segments

The total execution time of $testProgram$ that is represented with equation 56 can be derived by using the equations 510 and 59 :

$$C(u,v) = H(F^{(-1)}(u), G^{(-1)}(v)) = P[X \leq x, Y \leq y] \tag{511}$$

In order to construct the $P[X \leq x, Y \leq y]$ distribution, inverse CDFs $F^{(-1)}$, $G^{(-1)}$ must be found. $F^{(-1)}$ and $G^{(-1)}$ represents the inverse CDFs of the random variables X and Y in which they represent the execution time of the $testProgram_{self}$ and $testProgram_{sub}$, respectively. Obviously CDFs of the individual scopes are needed.

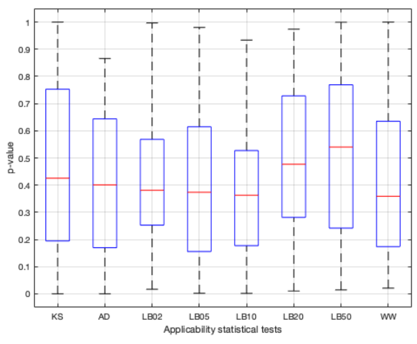
It is possible to model any random variable with a parametric or non-parametric probability distribution. However, modelling a random variable with a known parametric distribution based on the observations is not always possible. Furthermore, random variables might be composed of several sub random variables as in $testProgram_{self}$ and $testProgram_{sub}$. In such cases, modelling them by using their end-to-end measurement observations would result in ignoring both extreme cases and the cases resulting from the hybrid analysis. The idea is to benefit as much as possible from the observations in each scope. On the other hand, it is always possible to model the $testProgram$ function by using its end-to-end measurements as in chapter 3, but that is not the case in hybrid analysis. For the situations where modelling a random variable with a known parametric extreme value distribution is not possible, we propose to use empirical CDFs instead at the expense of resolution loss and incapability of prediction of capturing possible rare events.

The highest priority in our approach is to model each random variable with a parametric Extreme Value Distribution in order to predict the possible upcoming rare events based on the precise tail model. The exception is when the random variable is composed of several sub parts. In such a case, the scope of that random variable is analyzed based on our proposed method and finally an empirical CDF is derived for that block.

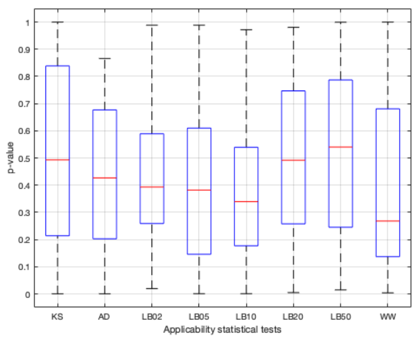
In accordance with the explanation given above, $testProgram_{self}$ and

$testProgram_{sub}$ parts of the $testProgram$ should not be modeled via an Extreme Value Distribution since they are composed of several sub-parts. It is instinctively known that parts of the $testProgram_{self}$ are not suitable to model with any type of Extreme Value Distribution since in reality there are no operations between the function calls and their execution time contribution is negligible. For these reasons and not to complicate the overall procedure, parts of the $testProgram_{self}$ are not tried to fit to any parametric continuous distribution. Hence, their empirical CDFs are constructed out of the historical observations. However, in reality these parts should also be tried to be modelled with an EV distribution.

On the other hand, the sub-scopes of $testProgram_{sub}$ can be modeled by a parametric EV distribution since they are not composed of sub-scopes meaning that they both do not have another function call inside. The primary requirement of EVT is to derive whether the original data conforms i.i.d assumption. The results of the statistical tests for $insertSortAsc$ and $insertSortDesc$ are given in Figure 5.5.



(a) insertSortAsc



(b) insertSortDesc

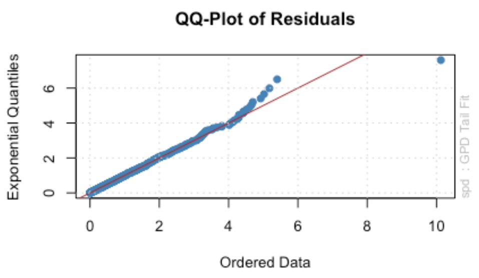
Figure 5.5: Statistical Test Results of the $testProgram_{sub}$ Segments

Results show that both X and Y in equation 58 are suitable to be modelled with one of the EV distributions. X and Y can either fit to a GEV or GPD based on the data selection method. In chapter 3, all programs were modeled via GEV because only the rightmost tails were important to derive the possible worst rare cases. However, in this case lower tails are also significant since there might be countercomonotonic situations where GEV would result in overestimation. Hence, in the proposed method we chose to model each random variable with Semi Parametric Piecewise Distribution which models each tail with GPD. In order to fit a Piecewise GPD distribution to the data, `spdfit` function of *spd* package is used in R environment [86]. Below is shown the fitting procedure in R which fits a Generalized Pareto Distribution to both upper and lower 10% tails of the distribution and fits a kernel distribution to the internals.

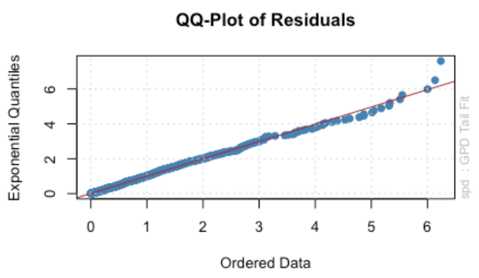
Listing 5.8: SPD Fitting Process for the Observations

```
> tails_insertSortAsc <- spdfit(insertSortAsc, upper = 0.9, lower = 0.1)
> tails_insertSortDesc <- spdfit(insertSortDesc, upper = 0.9, lower = 0.1)
```

QQ-Plots of the fitted distributions are given below.



(a) insertSortAsc



(b) insertSortDesc

Figure 5.6: QQ-Plots of the Fitted Piecewise GPD to $testProgram_{sub}$ Segments

Plots in Figure 5.6 show that the fitted distributions are suitable enough to represent the whole observation data of the $insertSortAsc$ and $insertSortDesc$. Thus, `qspd` function of *spd* package serves to represent the $F^{(-1)}$ and $G^{(-1)}$ of X and Y for $testProgram_{sub}$.

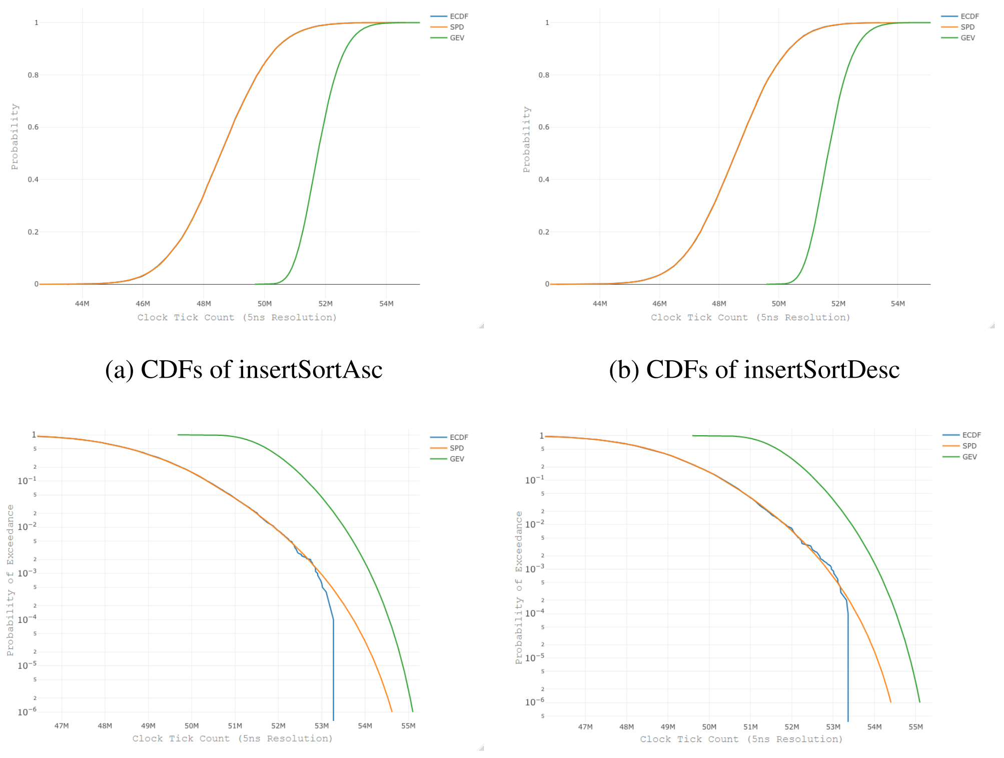
Therefore, reverting back to Figure 5.4c, possible next execution time of $testProgram_{sub}$ can be calculated as:

$$W_1(testProgram_{sub}) = F^{(-1)}(0.3318784754) + G^{(-1)}(0.675288088) = 97151546 \quad (512)$$

where 97151546 is the clock tick count and it represents one possible outcome for

$J(t) = P[X + Y \leq 97151546]$. Deriving the full $J(t)$ distribution is computationally intractable so the proposed method is used instead. In order to approximate the $J(t)$ distribution, samples must be generated as much as possible from the copula model. Applying the methodology in equation 512 to the whole matrix that is shown in Figure 5.4c results in an approximate distribution of $J(t)$ by combining the Extreme Value Theory and Copulas.

Each X and Y random variable that represents the execution time of $insertSortAsc$ and $insertSortDesc$ respectively are modeled with an Extreme Value distribution in order to capture the rare cases instead of using empirical CDF that is constructed from the historical observations. Benefits of this approach are discussed in chapter 3 and CDF plots for $insertSortAsc$ and $insertSortDesc$ are shown in the following figure.



(a) CDFs of insertSortAsc

(b) CDFs of insertSortDesc

(c) CCDFs of insertSortAsc

(d) CCDFs of insertSortDesc

Figure 5.7: CDF Plots of X and Y

Figure 5.7a and 5.7b shows the empirical CDFs of the functions based on observations and their model CDFs based on the fitted SPD and GEV distributions. As can be seen from the plots in the first row SPD models the whole distribution while GEV only targets the upper tail and overestimates the lower tails. From the Figures 5.7c and 5.7d which shows the Complementary CDF (CCDF) of both functions in logarithmic scale GEV safely upper bounds the whole distribution, but SPD fails to do so in some points. However, SPD still provides safe results for the possible extreme cases when the probability of exceedance is lower than 10^{-4}.

5.3.2.1.5 Estimating the Total Distribution

The equations to derive the total execution time distribution for $testProgram$ becomes:

$$W_i(testProgram_{sub}) = F^{(-1)}(u_i) + G^{(-1)}(v_i) \tag{513}$$

where:

i: $1, ..., 1.000.000$

$W_i(testProgram_{sub})$: is the execution time distribution of $testProgram_{sub}$

$F^{(-1)}$: is the inverse CDF of fitted SPD model of X in eq. 58

$G^{(-1)}$: is the inverse CDF of fitted SPD model of Y in eq. 58

u_i: are the simulated uniforms from the copula for $insertSortAsc$

v_i: are the simulated uniforms from the copula for $insertSortDesc$

$$W_i(testProgram_{self}) = F^{(-1)}(u_{(i)}) + G^{(-1)}(v_{(i)}) + M^{(-1)}(k_{(i)}) \tag{514}$$

where:

i: $1, ..., 1.000.000$

$W_i(testProgram_{self})$: is the execution time distribution of $testProgram_{self}$

$F^{(-1)}$: is the inverse ECDF of X in eq. 57

$G^{(-1)}$: is the inverse ECDF of Y in eq. 57

$M^{(-1)}$: is the inverse ECDF of Z in eq. 57

$u_{(i)}$: are the simulated uniforms from the copula for $testProgram_{entry}$

$v_{(i)}$: are the simulated uniforms from the copula for $insertSortAsc_{ret}$

$k_{(i)}$: are the simulated uniforms from the copula for $insertSortDesc_{ret}$

Finally,

$$W_i(testProgram) = F^{(-1)}(u_i) + G^{(-1)}(v_i) \tag{515}$$

where:

i: $1, ..., 1.000.000$

$W_i(testProgram)$: is the execution time distribution of $testProgram$

$F^{(-1)}$: is the inverse of $W(testProgram_{sub})$

$G^{(-1)}$: is the inverse of $W(testProgram_{self})$

u_i: are the simulated uniforms from the copula for $testProgram_{self}$

v_i: are the simulated uniforms from the copula for $testProgram_{sub}$

Note that F, G and M in equation 514 and F and G in equation 515 are step functions so they do not have unique inverse functions. Therefore, their *empirical quantile functions* are defined as the right inverse of the CDFs. Hence the quantiles are fetched from the inverse CDFs with the help of `findInterval` function of R.

Evaluating the calculations given in equations 513, 514 and 515 results in the following distribution which is compared with the observed end-to-end (EE) measurements, independent case and comonotonic case that is implemented in RapiTime tool.

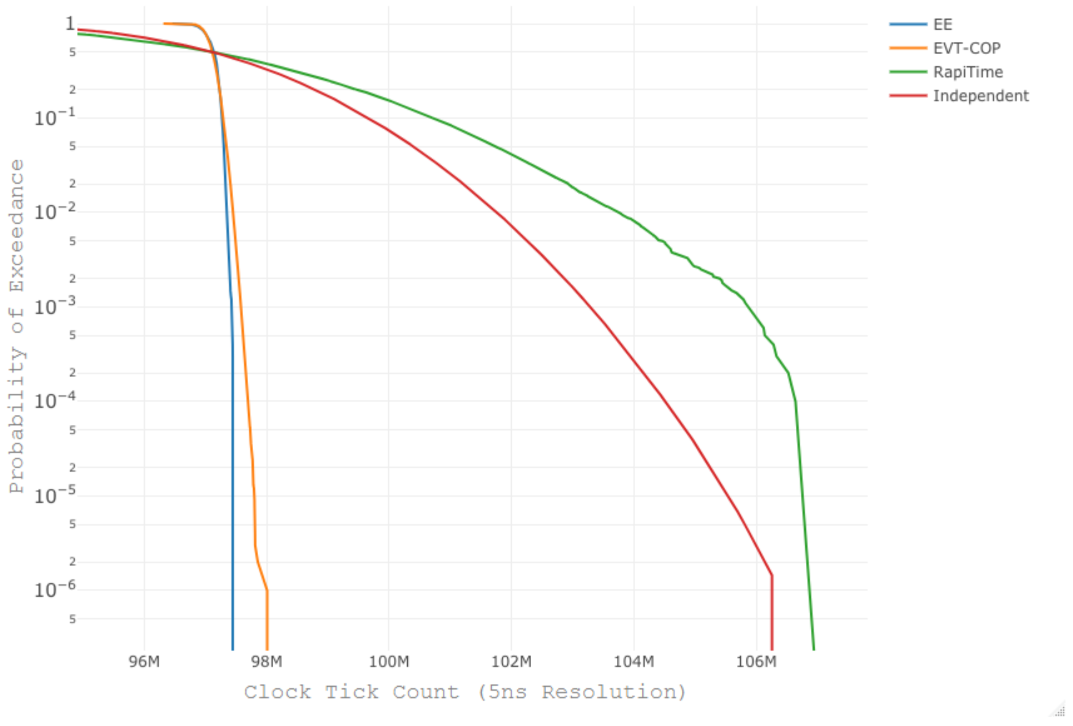


Figure 5.8: Estimated pWCET Distributions for $testProgram$

Following table summarizes some of the pWCET values in millisecond format that are obtained from the estimated distributions.

Table 5.2: Calculated pWCET Values for $testProgram$

pWCET	EVT-Cop	Independent	RapiTime	EE
pWCET(10^{-2})	487.2583 ms	508.8204 ms	518.8532 ms	486.7725 ms
pWCET(10^{-4})	488.4709 ms	522.6101 ms	539.0927 ms	487.3415 ms
pWCET(10^{-6})	490.0468 ms	539.0927 ms	539.0927 ms	487.3415 ms
pWCET(10^{-9})	490.0469 ms	539.0927 ms	539.0927 ms	487.3415 ms

Clearly, the biased convolution method (comonotonic assumption) and the standard convolution method (independent assumption) overestimate the results by a factor of 10% in the worst case. On the other hand, our approach based on EVT and Copulas results in a much tighter upper bound for the overall $testProgram$. The overestimation factor of 10% would not seem to be a significant problem, but this only happened for a very simple program that has only 2 consecutive function calls inside. That factor increases rapidly when the program structure is composed of more complex blocks such as iterative and conditional blocks. In the following sections we evaluated and compared our approach by analyzing more realistic programs.

5.4 Case Study - 1

In the previous section, application procedure of our method is detailed with a basic program that mainly focuses on the sequential block case. As was stated before, programs are composed of sequential, iterative and conditional blocks. Hence, a more realistic program should be composed of all types of blocks.
In this section, we constructed a synthetic and reproducible benchmark program in which the functions are selected from the Mälardalen WCET Benchmarks again [62]. The code is given below:

Listing 5.9: Synthetically Constructed Structured WCET Benchmark Program

```
void testProgram(void)
{
   float result = 0;
   int swapCnt = 0;
   vector vecA_tmp;

   vecA_tmp = f1(vec_A, 10); // select 10-th greatest element from vec_A

   swapCnt = f2(vecA_tmp); // Insert Sort

   f3(vec_A); // Insert Reverse Sort

   if (_scale > 50)
   {
      f4(vec_A, vec_C, vec_B, _scale); // FIR filter
   }
   else
   {
      f5(mat_A, mat_B, mat_C);    // Matrix Multiplication
   }

   int maxCnt = 100 - (int)(swapCnt/51);
   for (int i = 0; i < maxCnt; i++)
   {
      f6(vec_B[i]);   // Square-root
   }
}
```

The inputs vec_A, vec_B, mat_A and mat_B are randomly generated. In fact, vec_A and mat_A are the same objects, the only difference is their dimensions where

vec_A is 1 x 100 float array and mat_A is 10 x 10 float matrix. This is also valid for vec_B and mat_B. vec_C and mat_C are empty array and matrix that are used to store the results.

The program consists of 6 function calls and each of them depends on an input parameter which makes them multi-path programs. Additionally some of them depend on the output of the previous functions which creates a dependency between the execution time of the blocks. Briefly,

$f1$: corresponds to the `select` function which selects the Nth largest number in a floating point array. While doing that it quick sorts the array up to some point and returns the sorted array that is stored in $vecA\_tmp$

$f2$: is the same `insertSortAsc` function that is used previously

$f3$: is the `insertSortDesc`

$f4$: corresponds to the `fir` function which is a Finite impulse response filter over N items long sample. It consists inner loop with varying number of iterations and loop-iteration dependent decisions.

$f5$: corresponds to the `matmult` function which is a basic matrix multiplication function which has nested loops.

$f6$: corresponds to the `sqrt` function that is implemented by Taylor series.

After instrumenting the program, timing schema of the blocks and their corresponding IPoints are shown in the following expressions:

$$
\begin{aligned}
W(f1) &= W(IPoint_{27-26}) & W(f1_{ret}) &= W(IPoint_{26-25}) \\
W(f2) &= W(IPoint_{25-24}) & W(f2_{ret}) &= W(IPoint_{24-23}) \\
W(f3) &= W(IPoint_{23-22}) & W(f3_{ret}) &= max(W(IPoint_{22-21}), \\
& & & \quad W(IPoint_{22-19})) \\
W(f4) &= W(IPoint_{21-20}) & W(f4_{ret}) &= W(IPoint_{20-17}) \\
W(f5) &= W(IPoint_{19-18}) & W(f5_{ret}) &= W(IPoint_{18-17}) \\
W(f6) &= W(IPoint_{17-16}) & W(f6_{ret}) &= W(IPoint_{16-17})
\end{aligned}
$$

$$W(testProgram_{entry}) = W(IPoint_{29-27})$$

As in line with equation 56, the total execution time of this program can be represented as:

$$W(testProgram) = \underbrace{W(testProgram_{self})}_{\text{X}} + \underbrace{W(testProgram_{sub})}_{\text{Y}} \tag{516}$$

However, the expressions $W(testProgram_{self})$ and $W(testProgram_{sub})$ are not easily decomposed in this case as in equations 57 and 58. That is because we have conditional and iterative blocks that should be carefully examined.

In our proposed method, conditional and iterative blocks are handled as a whole scope on the highest level of the program scope. This approach is necessary in order to derive the copulas which models the dependency between the sub-scopes. This is best explained by the following expressions:

$$W(testProgram_{sub}) = \underbrace{W(f1)}_{\text{A}} + \underbrace{W(f2)}_{\text{B}} + \underbrace{W(f3)}_{\text{C}} + \underbrace{W(cond_{sub})}_{\text{D}} + \underbrace{W(loop_{sub})}_{\text{E}} \tag{517}$$

$$W(testProgram_{self}) = \underbrace{W(f1_{ret})}_{\text{A}} + \underbrace{W(f2_{ret})}_{\text{B}} + \underbrace{W(f3_{ret})}_{\text{C}} + \underbrace{W(cond_{self})}_{\text{D}} + \underbrace{W(loop_{self})}_{\text{E}} \tag{518}$$

$$W(cond_{sub}) = max(\underbrace{W(f4)}_{\text{A}}, \underbrace{W(f5)}_{\text{B}}) \tag{519}$$

$$W(cond_{self}) = max(\underbrace{W(f4_{ret})}_{\text{A}}, \underbrace{W(f5_{ret})}_{\text{B}}) \tag{520}$$

$$W(loop_{sub}) = \underbrace{\overbrace{W(f6)}^{\text{A}} + ... + W(f6)}_{\text{N}} \tag{521}$$

$$W(loop_{self}) = \underbrace{\overbrace{W(f6_{ret})}^{\text{A}} + ... + W(f6_{ret})}_{\text{N}} \tag{522}$$

Note that $W(cond_{sub})$, $W(cond_{self})$, $W(loop_{sub})$ and $W(loop_{self})$ expressions are same as the RapiTime approach [9]. However, while evaluating equations 517 and 518, directly replacing the conditional and iterative block expressions with their corresponding equations given in 519, 520, 521 and 522 would result in the same approach given in [9] which is the comonotonic assumption between the segments.

In our approach, instead of directly convolving all the expressions, a copula is fitted for the sub-scopes of $testProgram_{sub}$ and $testProgram_{self}$. After the simulation phase, the total execution time distribution is calculated by using the derived CDFs for the sub-scopes. Derivation of the CDFs for the sub-scopes are done in the same way as in the previous section for sequential blocks (A, B and C). However, for iterative and conditional blocks a special mechanism is developed.

In order to derive the copula for the sub-scopes of $testProgram_{sub}$ and $testProgram_{self}$, observations of the scopes need to be constructed. A, B and C in equation 517 are constructed with end-to-end measurements of $f1$, $f2$ and $f3$. For conditional and iterative blocks, their corresponding random variables are represented as:

$$D_i = (Obs_i(f4) \text{ or } Obs_i(f5)) \tag{523}$$

$$E_i = \sum_{j=1}^{N_i} Obs_j(f6),\ N_i = \text{current iteration count} \tag{524}$$

where: (525)

i = 1,..,10000

$Obs_i(f)$ = ith observed sample of function f

After constructing A, B, C, D and E for $testProgram_{sub}$, correlation between the pseudo-observations (u, v, k, m, n respectively) of the sub-scopes are shown in Figure 5.9.

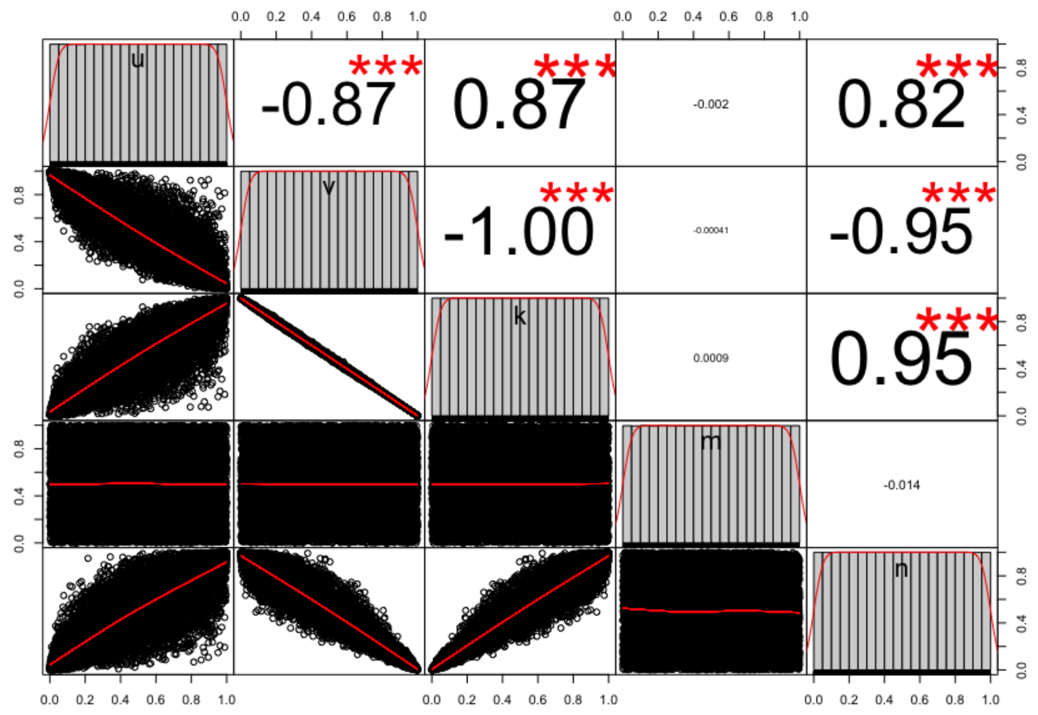
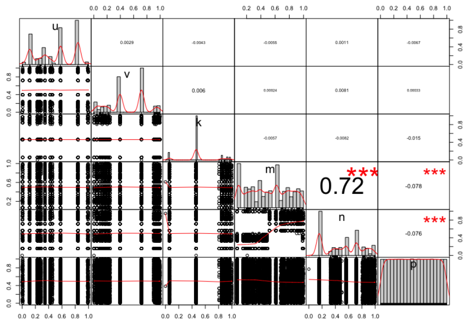
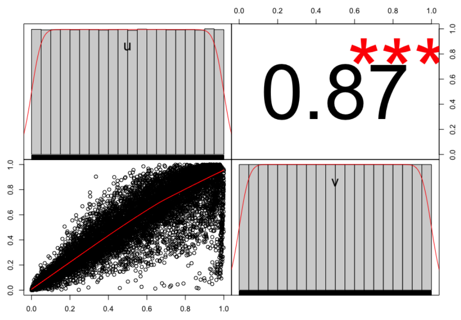


Figure 5.9: Correlation Plot of $testProgram_{sub}$

Same procedures are applied to $testProgram_{self}$ and $testProgram$ and the correlation between their scopes are shown in Figure 5.10.



(a) Correlation of $testProgram_{self}$



(b) Correlation of $testProgram$

Figure 5.10: Correlation Plots of the Program Scopes

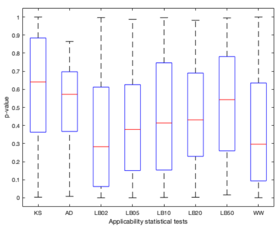
A Vine Copula model is fitted to each $testProgram$, $testProgram_{self}$ and $testProgram_{sub}$ scope by using the `RVineStructureSelect` function of R. After that goodness-of-fit test is applied to each copula with `RVineGofTest` function all the fitted copulas turned out to be valid. For simplicity of this section, results are not shown, but all the codes are provided.

The challenging part is the derivation of a CDF for each sub-scope of the scopes. For sequential sub-scopes which correspond to A, B, and C inside the $textProgram_{sub}$, same procedure can be applied as in previous section.

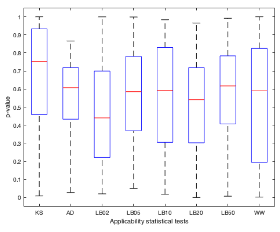
However, for conditional blocks, each part of the condition should be considered as a separate scope and our hybrid approach should be applied as they are separate programs. The result would be a distribution for each condition block and by using their inverses, equation 519 or 520 can be calculated. In this case study, condition blocks only consist of one functional block so it is valid to model them with SPD method if they conform the requirements.

For iterative blocks, their CDF is calculated by the approach that is detailed in [10] which is based on checking whether there is an independency across iterations. If so, then standard convolution is applied. Otherwise, the biased convolution is applied for equations 521 and 522. Instead of modelling each random variable inside the loop with their empirical CDFs (ETP), our method primarily aims to model them with SPD if possible.

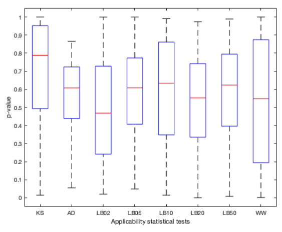
All functions inside the $testProgram$ conform the i.i.d requirements as shown in Figure 5.11.



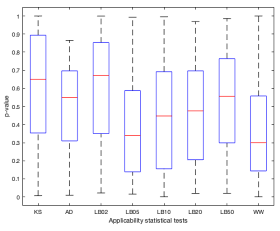
(a) f1



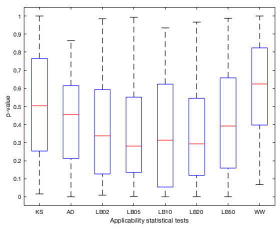
(b) f2



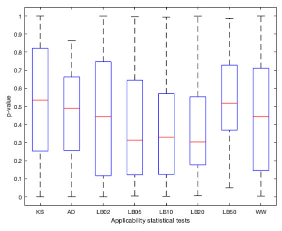
(c) f3



(d) f4

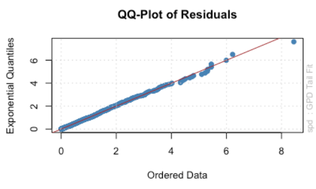


(e) f5

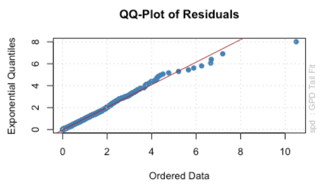


(f) f6

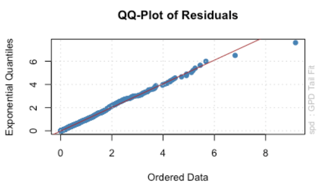
Figure 5.11: Statistical Tests of the Sub-Scopes of $testProgram_{sub}$



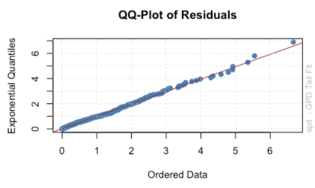
(a) f1



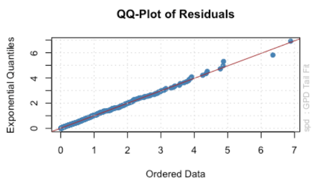
(b) f2



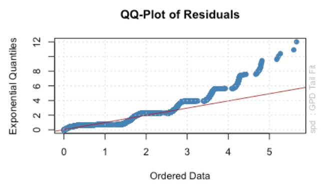
(c) f3



(d) f4



(e) f5

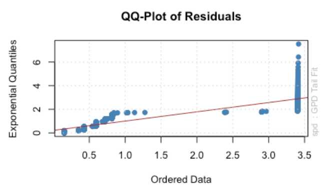


(f) f6

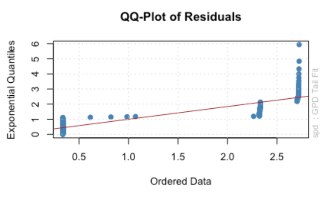
Figure 5.12: QQ-Plots of Piecewise GPD Fit for $testProgram_{sub}$

It is clearly seen from the Figure 5.12 that all the functions except $f6$ are suitable to be represented by an SPD model. According to our proposed method, if any random variable cannot be represented by an Extreme Value distribution, its empirical distribution would be used. For that reason empirical CDF will be used for $f6$ case only.

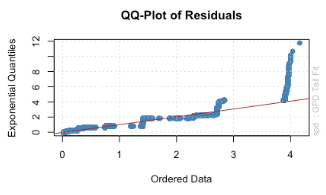
Sub-scopes of the $testProgram_{self}$ are not suitable to be modeled by an EV distribution as shown in Figure 5.13. Note that the QQ results for $f2$, $f3$ and $f5$ are missing because the `spdfit` function of R could not even estimate suitable parameters for them. Therefore, empirical CDFs are used to represent the random variables inside $testProgram_{self}$.



(a) f1



(b) f4



(c) f6

Figure 5.13: QQ-Plots of Piecewise GPD Fit for $testProgram_{self}$

Finally, the equations to derive the total execution time distribution for this case study becomes:

$$W_i(cond_{sub}) = max(F_A^{(-1)}(m_i), F_B^{(-1)}(m_i)) \tag{526}$$

where:

i: $1, ..., 1.000.000$

$W_i(cond_{sub})$: is the execution time distribution of $cond_{sub}$

$F_A^{(-1)}$: is the inverse CDF of fitted SPD model of A in eq. 519

$F_B^{(-1)}$: is the inverse CDF of fitted SPD model of B in eq. 519

m_i: are the simulated uniforms from the copula for D in eq. 517

$$F_{loop_{sub}} = \underbrace{F_A \circledast \cdots \circledast F_A}_{N} = F_A^{\circledast N} \tag{527}$$

$$W_i(loop_{sub}) = F^{(-1)}_{loop_{sub}}(n_i) \tag{528}$$

where:

i: $1, ..., 1.000.000$

$F_{loop_{sub}}$: is the CDF of E in eq. 517

F_A: is the ECDF of A in eq. 521

N: is the maximum observed iteration count

$\circledast$: is the convolution operation (standard or biased) for CDFs

$W_i(loop_{sub})$: is the execution time distribution of $loop_{sub}$

n_i: are the simulated uniforms from the copula for E in eq. 517

$$W_i(testProgram_{sub}) = \\ F_A^{(-1)}(u_i) + F_B^{(-1)}(v_i) + F_C^{(-1)}(k_i) + W_i(cond_{sub}) + W_i(loop_{sub}) \tag{529}$$

where:

i: $1, ..., 1.000.000$

$W_i(testProgram_{sub})$: is the execution time distribution of $testProgram_{sub}$

$F_A^{(-1)}$: is the inverse CDF of fitted SPD model of A in eq. 517

$F_B^{(-1)}$: is the inverse CDF of fitted SPD model of B in eq. 517

$F_C^{(-1)}$: is the inverse CDF of fitted SPD model of C in eq. 517

$W_i(cond_{sub})$: is the execution time distribution of $cond_{sub}$ given in eq. 526

$W_i(loop_{sub})$: is the execution time distribution of $loop_{sub}$ given in eq. 528

u_i: are the simulated uniforms from the copula for A in eq. 517

v_i: are the simulated uniforms from the copula for B in eq. 517

k_i: are the simulated uniforms from the copula for C in eq. 517

Similar procedure is applied for $testProgram_{self}$ in eq. 518 and its total CDF is derived. The only difference is that ECDFs are used to represent the segments because of the unsuitability of SPD for modelling them as shown in Figure 5.13.

Note that the convolution operation used for loop case is the standard convolution. The correlation between the iteration index parameter and the execution time of $f6$ is shown in Figure 5.14.

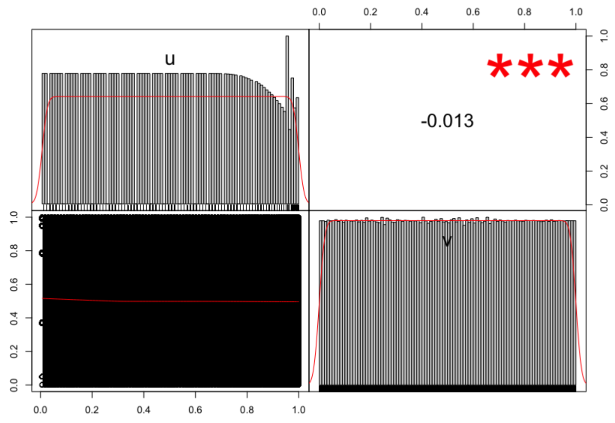


Figure 5.14: Correlation Across the Iterations

u represents pseudo-observations of loop iteration index parameter and v represents pseudo-observations of execution time of $f6$. It can be seen that there is no correlation between iteration index and execution time of $f6$. It means that the execution time of $f6$ does not increase when loop iteration count increases or vice-versa. In order to calculate eq. 521 standard convolution of random variable A is sufficient. This calculation is basically sum of N random variables which is the main problem of our work. Therefore, the same copula approach could be applied, but in order to do that $1.000.000$ x N simulated uniforms should have been generated with a fitted copula. Then by using inverse CDF of $f6$, eq. 521 could have been calculated as $testProgram$ itself. However, this approach is computationally intractable so for loop blocks an adequate independence test is sufficient to decide whether to use standard convolution or biased convolution.

Finally, solving the same equation shown in eq. 515 yields Figure 5.15:

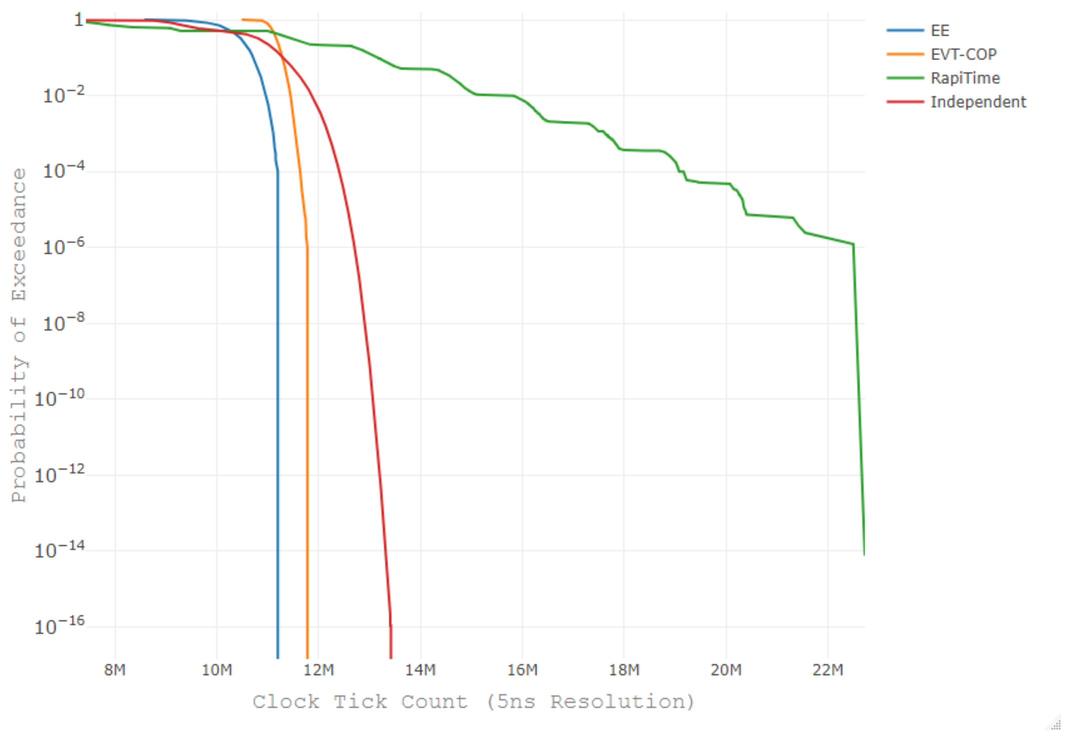


Figure 5.15: Estimated pWCET Distributions for Case-Study 1

Our main result is shown in Figure 5.15 which is tightest among all methods. Although the standard convolution is used to calculate the execution time distribution of the loop block, it still stays below the independent method. It can be clearly seen how the comonotonic assumption of commercial RapiTime tool overestimates the results by a huge factor.

Following table summarizes some pWCET values that are obtained from the estimated distributions.

Table 5.3: Calculated pWCET Values for Case-Study 1

pWCET	EVT-Cop	Independent	RapiTime	EE
pWCET(10^{-2})	57.272 ms	60.219 63 ms	79.1122 ms	54.848 87 ms
pWCET(10^{-4})	58.195 58 ms	63.757 11 ms	95.840 14 ms	56.323 95 ms
pWCET(10^{-6})	58.678 75 ms	67.016 77 ms	113.6355 ms	56.323 95 ms
pWCET(10^{-9})	58.709 44 ms	68.353 41 ms	113.6355 ms	56.323 95 ms

Results in Table 5.3 show that the RapiTime's approach overestimates the observed end-to-end execution time more than 100% for pWCET(10^{-9}) case. This ratio is unacceptable for real-time embedded software environments which have limited scheduling resources. However, our results are both safe based on the tail modelling methodology (EVT) and tight by the help of proposed dependency modelling approach (Copulas).

5.4.1 Tightness Assessment

In order to provide the evidence of tightness and reliability of the proposed EVT-Copula method, 10^6 additional observations are collected from the same setup. The maximum observed execution time out of the 10^6 measurements for the Case Study - 1 was 57.44 ms. It shows that our EVT-Copula method is both tight and reliable compared to the Independent and Comonotonic assumptions.

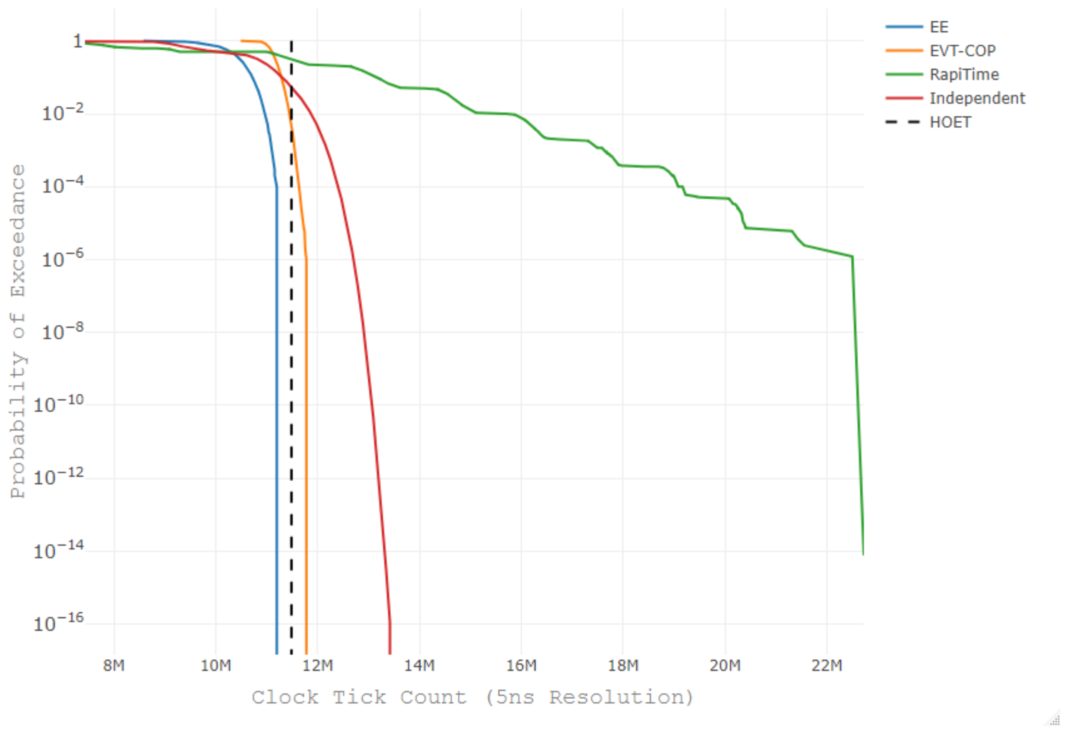


Figure 5.16: Tightness Assessment Results for Case-Study 1

Figure 5.16 shows $HOET$ of 10^6 observations in black dashed line. It can be clearly seen that our EVT-Copula method is both safe and tightest for pWCET(10^{-6}) while the RapiTime's approach overestimates by a huge factor.

5.5 Case Study - 2

Our last case study is the analysis of an industrially developed software component. Because of the confidentiality reasons, the original code of the component function is not given. However, its pseudo-code which clarifies the structure of the analyzed function is given.

The analyzed software component is a function of Satellite Flight Control software that is developed in Turkish Aerospace Industries, Inc. (TAI) which is developed by conforming to the requirements defined by ECSS-E-ST-70-41C - Packet Utilization Standard(PUS) by European Space Agency (ESA) [87]. Each service defined inside the standard is developed as a separate component and our aim is to analyze a function of Memory Management (MEMMAN) service which provides the capability for loading, dumping and checking the contents of the memories that are present on-board.

Most of the functionality of the services are triggered by the received telecommands from the ground control station. MEMMAN service has lots of capabilities, which can be triggered by different commands and each of the received commands are processed by a different function. From the beginning of the function call until its return specifies the response time for that command.

The relation between the execution time and the response time is out of the scope of this study. However, in our case the response time is equal to the execution time of the telecommand processing function. In this case study we analyze the processing function of *Load Raw Memory Data Areas (TC[6,2])* telecommand which commands to write the specified raw data to specified memory on-board. The telecommand basically contains the memory ID indicating which memory the data is to be written and a list of raw data. The structure of the telecommand and the generated telemetries by the onboard computer conforms to the requirements defined in PUS standard. Based on the obtained parameters, the processing function performs the task in accordance with the requirements. The pseudo-code of the analyzed component function is given below:

Listing 5.10: Industrially Developed Function of a Software Component

```
void process_loadRawMemoryDataAreas(Request telecommand)
{
  failCode = isRequestValid(telecommand); //telecommand validation

  if(failCode == NO_ERROR)
  {
    send_tm_1_3(telecommand); //Send request acceptance ack telemetry

    for(int i = 0; i < telecommand->N; i++)
    {
      writtenData = writeToMemory(telecommand->memoryID, telecommand->
          rawData[i]); //Write raw data to memory

      calculatedCRC = calculateCRC(writtenData); //Calculate the CRC of
          written data

      if(calculatedCRC != telecommand->CRC[i])
      {
        send_tm_1_4(telecommand); //Send fail telemetry
        goto exit;
      }
    }

    send_tm_1_7(telecommand); //Send successful completion telemetry
  }
  else
  {
    send_tm_1_4(telecommand); //Send fail telemetry
  }

  exit: return;
}
```

The same test bench is used to analyze the function which runs the function 10000 times with randomly generated $telecommand$ input parameter.

Received telecommand message is first checked whether the parameters inside are valid or not. If the parameters are validated then the process continues, otherwise a telemetry report is generated and sent to ground to indicate that the request is rejected.

After the request is accepted, a telemetry message is sent to ground to inform that the received request message is accepted. The main processing task is done inside the loop that iteratively writes the data to the given memory and after the writing process finishes, an integrity check is done in order to make sure that written data is not corrupted during the writing process. If any error happens than a fail report is generated to be sent to ground while terminating the processing task.

Because we generated the $telecommand$ message randomly, the function sometimes accepted the request, but sometimes not. When the request is accepted, the data is written to a memory which is randomly selected (randomly generated memory ID parameter for the $telecommand$ message). Furthermore, the N parameter inside the $telecommand$ message is also randomly generated which makes the loop iterates dependent to the given $telecommand$ message.

The expressions to derive the execution time distribution are given as:

$$W(testProgram) = \underbrace{W(testProgram_{self})}_{\text{A}} + \underbrace{W(testProgram_{sub})}_{\text{B}} \tag{530}$$

$$W(testProgram_{sub}) = \underbrace{W(f1)}_{\text{A}} + \underbrace{W(cond1_{sub})}_{\text{B}} \tag{531}$$

$$W(testProgram_{self}) = \underbrace{W(testProgram_{entry})}_{\text{A}} + \underbrace{W(f1_{ret})}_{\text{B}} + \underbrace{W(cond1_{self})}_{\text{C}} \tag{532}$$

$$W(cond1_{sub}) = max(\underbrace{W(cond1_{if})}_{\text{A}}, \underbrace{W(f6)}_{\text{B}}) \tag{533}$$

$$W(cond1_{self}) = max(\underbrace{W(cond1_{if_{ret}})}_{\text{A}}, \underbrace{W(f6_{ret})}_{\text{B}}) \tag{534}$$

$$W(cond1_{if}) = \underbrace{W(f2)}_{\text{A}} + \underbrace{W(loop_{sub})}_{\text{B}} + \underbrace{W(f5)}_{\text{C}} \tag{535}$$

$$W(cond1_{if_{ret}}) = \underbrace{W(f2_{ret})}_{\text{A}} + \underbrace{W(loop_{self})}_{\text{B}} + \underbrace{W(f5_{ret})}_{\text{C}} \tag{536}$$

$$W(loop_{sub}) = \underbrace{\overbrace{W(loop_{body})}^{\text{A}} + ... + W(loop_{body})}_{\text{N}} \tag{537}$$

$$W(loop_{self}) = \underbrace{\overbrace{W(loop_{body_{ret}})}^{\text{A}} + ... + W(loop_{body_{ret}})}_{\text{N}} \tag{538}$$

$$W(loop_{body}) = \underbrace{W(f3)}_{\text{A}} + \underbrace{W(f4)}_{\text{B}} + \underbrace{W(cond2_{sub})}_{\text{C}} \tag{539}$$

$$W(loop_{body_{ret}}) = \underbrace{W(f3_{ret})}_{\text{A}} + \underbrace{W(f4_{ret})}_{\text{B}} + \underbrace{W(cond2_{self})}_{\text{C}} \tag{540}$$

$$W(cond2_{sub}) = max(\underbrace{W(f6)}_{\text{A}}, 0) \tag{541}$$

$$W(cond2_{self}) = max(\underbrace{W(f6_{ret})}_{\text{A}}, 0) \tag{542}$$

Where:

$f1$: represents the `isRequestValid` function

$f2$: represents the `send_tm_1_3` function

$f3$: represents the `writeToMemory` function

$f4$: represents the `calculateCRC` function

$f5$: represents the `send_tm_1_7` function

$f6$: represents the `send_tm_1_4` function

It can be seen from the above equations and the pseudo-code that $f6$ is called from two different scopes. However, this does not affect the derivation of $W(f6)$ and $W(f6_{ret})$. They are used either in eq. 533, 534 or 541, 542 and the same derived CDFs for $f6$ and $f6_{ret}$ are used for both scopes.

Unfortunately, in our experiments no data corruption is observed during the writing phase. In fact, that conditional control is only coded for defensive reasons. Therefore, `send_tm_1_4` function is not called from the scope of $W(cond2_{sub})$. On the other hand, the same function has been called from the scope of $W(cond1_{sub})$, thus $W(f6)$ and $W(f6_{ret})$ exists. Theoretically it is possible to derive the $W(cond2_{sub})$ and $W(cond2_{self})$, but deriving the CDFs of those scopes are not necessary. This can be clarified by adapting equation 523 to random variables C inside equations 539 and 540. It can be seen from the results that random variables representing the observed values for $f6$ and $f6_{ret}$ under the scope of $W(cond2_{sub})$ and $W(cond2_{self})$ are equal to 0. Hence, it is not possible to fit a copula for $W(cond2_{sub})$ and $W(cond2_{self})$ inside equations 539 and 540. Therefore, $W(cond2_{sub})$ and $W(cond2_{self})$ parameters are assumed to be neutral elements and are extracted from the equations.

Following the same procedures as before results in the following pWCET distribution.

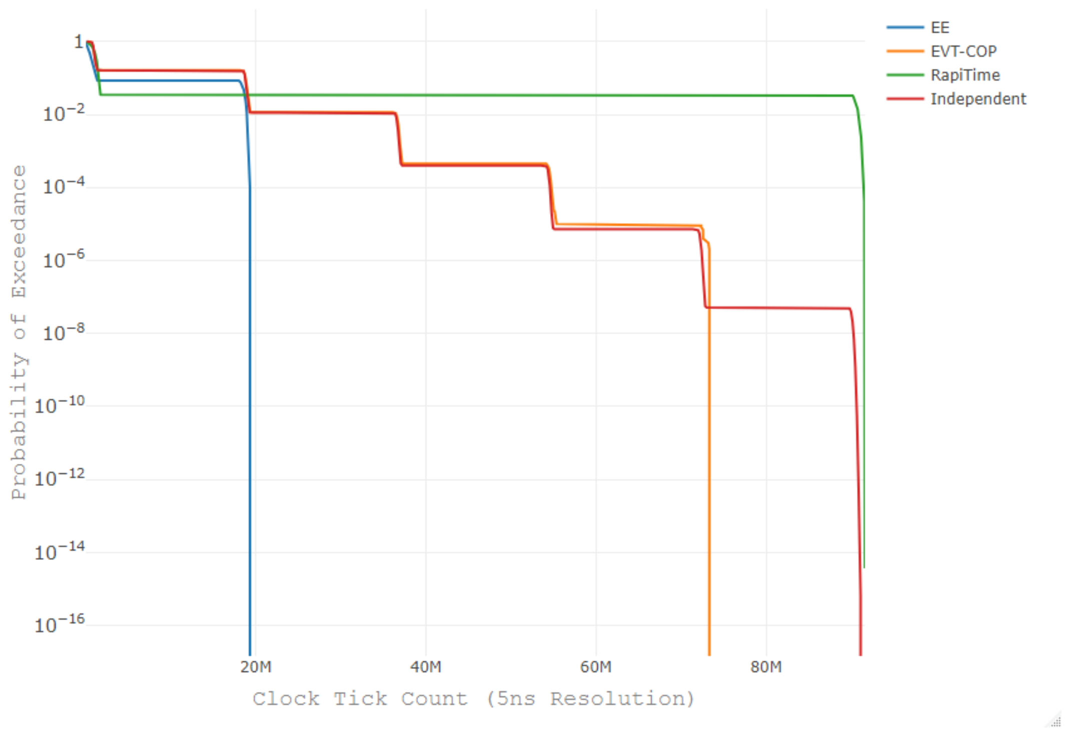
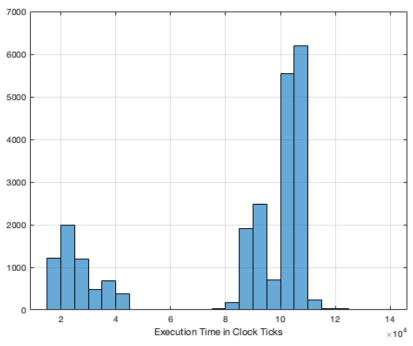


Figure 5.17: Estimated pWCET Distributions for Case-Study 2

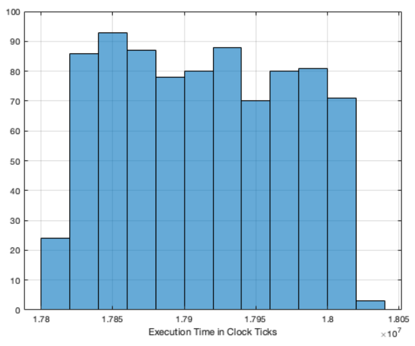
Table 5.4: Calculated pWCET Values for Case-Study 2

pWCET	EVT-Cop	Independent	RapiTime	EE
pWCET(10^{-2})	182.774 ms	182.1895 ms	454.5465 ms	94.758 63 ms
pWCET(10^{-4})	274.2195 ms	272.9592 ms	457.6076 ms	96.656 38 ms
pWCET(10^{-6})	364.5849 ms	362.497 ms	458.1172 ms	96.656 38 ms
pWCET(10^{-9})	364.6062 ms	452.5728 ms	458.1172 ms	96.656 38 ms

It can be seen from the results that all methods overestimated the WCET by a huge factor. The main reason behind this overestimation results from the derived CDF of $loop_{sub}$. $loop_{sub}$ is an iterative block which contains $f3$ that represents the `writeToMemory` function. `writeToMemory` function performs the main task of writing raw data to specified memory. As expected, writing data to a memory is an I/O operation which is excessively dependent to the given memory type and address. During the analysis runs it is observed that some calls to this function took too much time to finish. In fact, the following histogram plots summarizes the issue. The execution time of $f3$ is divided into two parts where the majority of the traces are laid between $(16.000, 140.000)$, but some of the traces are laid around $1.8x10^7$ clock ticks. It shows that for some parameters inside the $telecommand$, the execution time of $f3$ is 100 times more than the usual.



(a) $f3 \leq 10^6$



(b) $f3 > 10^6$

Figure 5.18: Histogram Plots of $f3$ of Case-Study 2

Combining this effect with an iteration leads to an overestimation up to 400% as shown in Table 5.4.

Nevertheless, our EVT-Copula method still stays below the state-of-the-art method of RapiTime's. It is interesting that our method follows almost the same path with the standard convolution method. This is because the fitted copulas for the scopes inside the analyzed program were either independent or gaussian copula with a parameter close to 0. This results in the independent behavior between the blocks inside a program, thus resulting a distribution which is very close to the independent case.

5.5.1 Tightness Assessment

As in line with the previous case study, we also collected 10^6 additional observations to provide evidence for tightness and reliability. The maximum observed execution time out of 10^6 measurements for Case Study - 2 was 133.4 ms. By comparing the pWCET(10^{-6}) values in Table 5.4 with the $HOET$ value, our EVT-Copula method is both safe and tight enough.

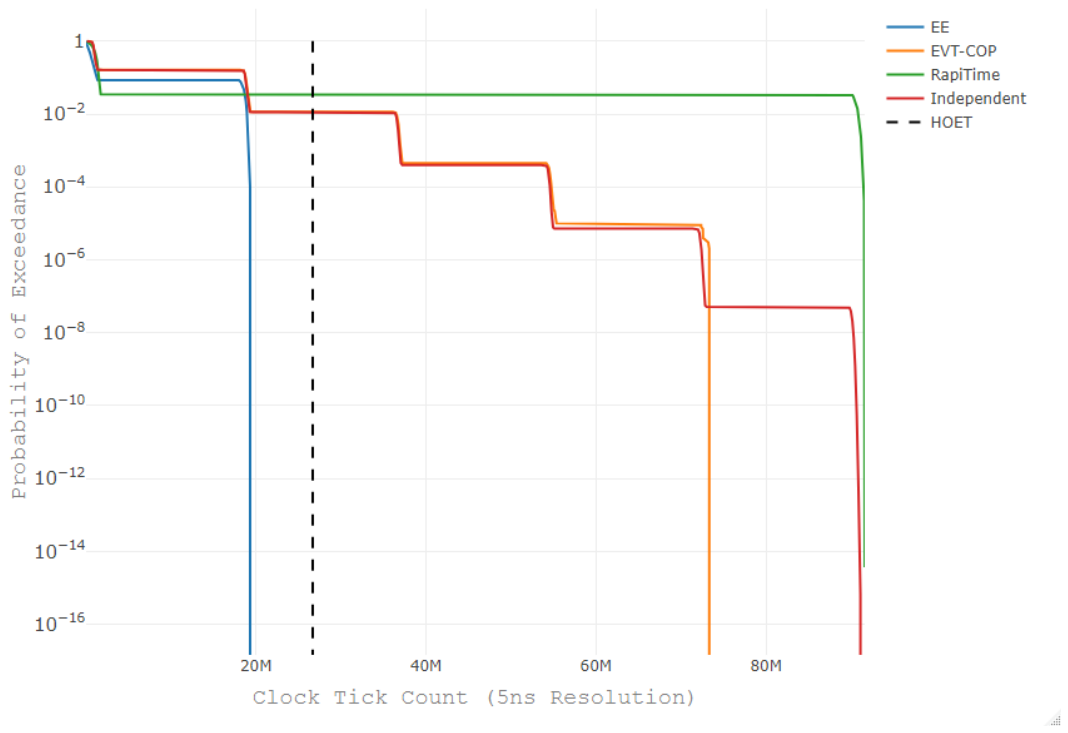


Figure 5.19: Tightness Assessment Results for Case-Study 2

Figure 5.19 shows the estimated pWCET values with the observed $HOET$ value out of 10^6 observations. It can be clearly seen that all methods safely upper bounds the $HOET$ value for pWCET(10^{-6}), but they still overestimate by a huge factor.

pWCET(10^{-9}) for the EVT-Copula may provide a tighter result, but it was not feasible to collect 10^9 observations from our system since collecting 10^6 measurements from the system already took 2 days.

5.6 Conclusion

In this section, we introduced our Hybrid Probabilistic Timing Analysis framework with EVT and Copulas for COTS platforms which is not only theoretical but also an industrially viable solution.

Current state-of-the-art hybrid probabilistic approach which is also implemented in a commercial tool named as RapiTime divides the programs into basic blocks and represents each basic block with their ETPs. Those ETPs are constructed with a frequentist approach where each possible execution time of the block is captured through measurements and their frequency of observations are assigned as their probabilities. After constructing all ETPs, a biased convolution is performed to estimate the overall pWCET distribution. However, this method is lack of capturing rare cases for individual blocks and it overestimates the results with comonotonic dependency assumptions through biased convolution.

What we propose is to divide the program into functional scopes and derive execution time distribution for each scope where they are eventually combined to construct the overall execution time distribution of the analyzed program. Copulas are used to model the dependency between the sub-scopes inside each scope. Then by using Monte-Carlo simulation approach, each scope is simulated through copulas and by the help of Extreme Value Theory extreme events are also captured.

Results of the experimental example and our case studies show that RapiTime's approach and independent assumption overestimates the results by a huge factor. Our method on the other hand provided the tightest results while capturing rare cases for each individual block inside the scopes.

Of course not all the blocks were suitable to be modelled via EVT. The blocks where the execution time variability was not sufficient or the variation across different scenarios conducted large steps in the distribution namely *mixed distributions* as detailed in [14] were unable to be modelled by an EV distribution. `writeToMemory` function in Case-Study 2 was an example to that. Modelling such distributions with EVT is out of the scope of this study, but our approach allows to use standard ETPs as a backup mechanism. This way, it was not possible to capture rare cases for those type of functions, but the computation of the overall distribution could proceed.

CHAPTER 6

RESULTS AND CONCLUSION

This thesis proposed an industrially-viable enhanced hybrid probabilistic timing analysis framework for time-critical applications running on top of a COTS hardware platform.

In Chapter 1, the motivation for the development of the proposed framework is given by noting the following key points:

1. Recent improvements in embedded processor technology compels the static timing analysis techniques which require to precisely model every aspect of the underlying platform. Consequently, lack of knowledge about the platform and unpredictabilities resulting from advanced architectural properties forces static methods to manage the complexities by making conservative assumptions which eventually leads to overestimation of the WCET results.

2. Jittery response times of complex hardware allows to apply widely used statistical methods for the WCET analysis in order to reduce the analysis cost compared to the conventional measurement-based timing analysis techniques which theoretically require the full path coverage to be reliable.

3. One of the existing statistical methods makes use of convolution operations to derive the overall probabilistic WCET distribution with a conservative convolution approach named as biased convolution. However, this approach overestimates the results and neglects the extreme cases for individual blocks.

4. The majority of the studies in statistical timing analysis domain utilize a time-randomized platform to focus on statistical methods and eliminate some of the

prerequisites of EVT. However, an industrially-viable methodology is needed to apply those techniques on top of a COTS platform.

Therefore, we proposed a framework which combines EVT and Copulas to decrease the overestimation while capturing the extreme cases.

In Chapter 2, we gave the basic notions of WCET analysis and current state-of-the-art techniques. A literature review revealed that there is no known study to this day which tried a hybrid approach with Copulas and EVT for the timing analysis of the real-time applications running on top of a COTS platform.

In the same chapter, application prerequisites of EVT and its application details for measurement-based timing analysis are given. We also highlighted the importance of source of execution time variability concept and mentioned that the only acceptable solution for COTS platforms is the random input selection for the programs. This property is also important for deciding the granularity of the blocks for which the execution times are collected. Because the only option is to use randomized inputs, the programs to be analyzed are divided into functional blocks which increases the possibility of EVT to be applicable, but decreases the available structural information of the analyzed program. However, dividing the program into functional granularity is also necessary to decrease the instrumentation overhead resulting from injecting probes throughout the source code.

In order to decrease the overestimation of the current state-of-the-art solution of RapiTime's, Copulas from statistics domain are utilized. Details of the copulas and their usage in timing analysis domain are also given in Chapter 2.

In Chapter 3, application details of the MBPTA with EVT on our COTS platform are given. We present the hardware details of our COTS platform which is a widely preferred solution in aerospace domain. We also introduce the software architecture and the scheduling mechanism conducted on our platform to clarify the necessity of WCETs of the software components.

Majority of MBPTA studies in the literature utilize custom-made randomized platforms which especially targets to randomize cache replacement and placement or bus arbitration policies. However, it is not always possible to construct such randomized platforms which are identical to the underlying COTS platform in industry. There are also software solutions to randomize the cache behavior to increase the applicability of EVT, but these also require specialized compilers or RTOS source code modifications. These actions are not easy to take in industrial applications. Therefore, the solutions for COTS platforms reduce to random input selection and cache flushing which we also employed in our framework. All the constraints and assumptions about the applicability of EVT on a COTS platform are given in the same chapter.

The test bench introduced in that chapter is used throughout this study for the timing analysis of the software components. In order to increase path coverage and make programs comply with EVT requirements, a random input generation mechanism is constructed. We also provide the basic steps to follow to apply EVT in COTS platforms. The outcomes of that chapter compose the assumptions and required steps to follow in chapter 5.

In Chapter 4, we introduced our mechanism to increase the confidence of the measurements by decreasing the probe effect along with custom developed TraceBox hardware. It is one of the building blocks of our analysis framework which allowed us to capture fine-grained execution time traces rather than end-to-end measurements of the software components.

The hardware is developed on an Altera Cyclone V based FPGA board along with an ARM processor. It is considered as a SoC FPGA board which allows to design low level logic inside the FPGA and management software on the ARM side. Developed trace capturing hardware is able to capture high speed GPIO signals, time stamp and store them on-board which theoretically can capture forever as long as the space inside the SDCard suffices. Results show that the proposed hardware&software mechanism decreased the probe effect by 98% compared to online storage technique which is a widely used software only approach.

In Chapter 5, the main hybrid probabilistic timing analysis framework is introduced. The motivation behind the necessity of an improvement over commercial measurement-based timing analysis tool RapiTime is discussed. Hence, the details of the proposed framework which is based on a methodology used in economics is adapted to the timing analysis domain.

The main principle behind the proposed framework is to model the dependency between the random variables with Copulas and model each random variable with a proper Extreme Value distribution whenever possible. This method is not strictly limited to our proposed definitions. In fact, the mechanism behind the commercial tool RapiTime is revealed and some improvements and modifications over their method are done in this study.

In reality, Copulas are also studied by the developers of RapiTime, but that study is only a demonstration to show that their conservative convolution method safely upper-bounds all possible cases. However, this approach overestimates the results tremendously as shown in our experimental evaluations.

Instead of representing the whole program as one syntax tree, a specialized version similar to the scope-tree approach is followed. In order to fit a copula for each scope of the program, this representation method was necessary. Basically, the program is divided into scopes and each scope is analyzed in isolation by using our proposed methodology.

The application procedure and constraints are detailed by experimental evaluations and case studies throughout the chapter. While deriving the CDFs for each segment, procedures defined in Chapter 3 are used and a type of Extreme Value distribution is fit to represent the random variables if possible.

Results of the experiments show that the proposed EVT with Copula method provides much tighter results than the comonotonic convolution or standard convolution methods. More importantly, all the experiments are done on our industrial COTS platform with real trace data which provides the evidence of applicability of this approach in the industry.

CHAPTER 7

FUTURE WORK

In this chapter, the future directions and possible extension points of this study are given.

First of all, the main aim of this study was to provide a hybrid probabilistic measurement - based timing analysis mechanism for COTS platforms. Because of this the only applicable source of the execution time variability is the random input generation. This randomization process can be made more intelligent such as using model checking or genetic algorithms as explained in [88, 89, 90, 52]. By eliminating the irrelevant inputs for the analyzed program, path coverage could be increased.

For the application of EVT, most of the procedures are based on visual checking from the plots in order to conclude that the models are fit to the data or the data conform the applicability requirements. In fact, there are recent studies which calculates the applicability of EVT numerically and selects the best distribution to represent the observed data such as MBPTA-CV [14]. Furthermore, there is also an open-source study which implemented the MBPTA-CV method in C++ environment named as *chronovise* [39]. By adapting more automatic mechanisms like these, the necessity of user interaction of the proposed framework could be decreased.

We used L-Moments approach to estimate the EVT model parameters in Chapter 3 in MATLAB environment for GEV which proved to outperform the MLE approach [16]. However, `spdfit` function of R which fits GPD to both tails of the distributions estimates the model parameters with MLE approach. Since it is possible to modify the source codes of the functions of R, `spdfit` can be modified to utilize L-Moments instead of MLE to increase the reliability.

On the other hand, instrumentation points are injected into the source code of the analyzed program. While the proposed mechanism minimizes the probe effect, injecting the probes in assembly level would decrease the probe effect much more. However, in order to do that a specialized compiler or linker support is needed.

The analysis phase in the proposed framework is done offline on a host computer after the test runs are finished. It is not possible to have an idea whether the collected trace data are enough or not to apply statistical methods or whether the coverage is sufficient or not. Therefore, a mechanism could be introduced to make the analysis online by connecting the TraceBox hardware and the host computer through a network and stream the trace data over the network interface from TraceBox to the host computer. This way, it would be possible to make sure that enough data is collected and termination criteria is met.

This thesis study presents a framework which is implemented in MATLAB and R which are mainly considered as prototyping environments. The experiments and case studies are analyzed mostly by manual modifications or rewritings of the analysis code in R or MATLAB. In order to transform this study into a full function and automatic tool, a number of improvements should be made as follows:

1. An automatic C/C++ source code parser is needed in order to extract the structural information of the analyzed code and instrument the necessary points automatically.
2. A trace parser is needed to extract the execution traces of each individual instrumentation point and associate them with the corresponding blocks inside the program.
3. Representation of the analyzed program with our proposed scope tree approach should be automatized.
4. A fully automatic mechanism to derive the copulas and estimate the extreme value distributions by testing the applicability is needed.
5. Instead of using Monte-Carlo approach which is basically simulating the scope behaviors, other numerical methods such as GAEP algorithm [80] could be used.

Last but not the least, all these capabilities should be transformed into a more efficient programming language such as C or C++.

REFERENCES

[1] R. Wilhelm, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, and R. Heckmann, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, pp. 1–53, apr 2008.

[2] F. J. Cazorla, L. Kosmidis, E. Mezzetti, C. Hernandez, J. Abella, and T. Vardanega, "Probabilistic Worst-Case Timing Analysis," *ACM Computing Surveys*, vol. 52, pp. 1–35, feb 2019.

[3] F. J. Cazorla, T. Vardanega, E. Quiñones, and J. Abella, "Upper-bounding Program Execution Time with Extreme Value Theory," *International Workshop on Worst-Case Execution Time Analysis 2013 (WCET'13)*, vol. 30, no. Wcet, pp. 64–76, 2013.

[4] A. Corporation, *DE0-Nano-SoC User Manual*, vol. 5. 2015.

[5] RTCA and EUROCAE, "Do-178c / ed-12c, software considerations in airborne systems and equipment certification," tech. rep., RTCA and EUROCAE, 2012.

[6] I. O. for Standardization, "Iso/dis 26262. road vehicles - functional safety," tech. rep., ISO, 2009.

[7] CENELEC, "En50128 railway applications: Software for railway control and protection," tech. rep., CENELEC, 2001.

[8] ECSS-Secretariat, "Ecss-e-st-40c, space engineering: Software," tech. rep., ESA-ESTEC Requirements and Standards Division, 2009.

[9] G. Bernat, A. Burns, and M. Newby, "Probabilistic timing analysis: An approach using copulas," *Journal of Embedded Computing*, vol. 1, p. 179, jun 2005.

[10] G. Bernat, A. Colin, and S. Petters, “WCET analysis of probabilistic hard real-time systems,” in *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pp. 279–288, IEEE Comput. Soc, 2002.

[11] K. Avdulaj, “Value-at-Risk based on Extreme Value Theory method and copulas . Empirical evidence from Central Europe .,” 2010.

[12] R. Wilhelm, “Determining bounds on execution times,” *Embedded Systems: Handbook*, pp. 14–1–14–23, 2005.

[13] F. Wartel, L. Kosmidis, C. Lo, B. Triquet, E. Quinones, J. Abella, A. Gogonel, A. Baldovin, E. Mezzetti, L. Cucu, T. Vardanega, and F. J. Cazorla, “Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study,” in *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, no. Sies, pp. 241–248, IEEE, jun 2013.

[14] J. Abella, M. Padilla, J. D. Castillo, and F. J. Cazorla, “Measurement-Based Worst-Case Execution Time Estimation Using the Coefficient of Variation,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 22, pp. 1–29, jun 2017.

[15] A. Burns and S. Edgar, “Predicting computation time for advanced processor architectures,” in *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*, pp. 89–96, IEEE Comput. Soc, 2000.

[16] K. P. Silva, L. F. Arcaro, and R. S. de Oliveira, “On Using GEV or Gumbel Models When Applying EVT for Probabilistic WCET Estimation,” in *2017 IEEE Real-Time Systems Symposium (RTSS)*, vol. 2018-Janua, pp. 220–230, IEEE, dec 2017.

[17] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones, and F. J. Cazorla, “Measurement-based probabilistic timing analysis for multi-path programs,” in *Proceedings - Euromicro Conference on Real-Time Systems*, pp. 91–101, IEEE, jul 2012.

[18] B. Y. A. Ferreira and L. De Haan, “On the block maxima method in extreme value theory: PWM estimators,” *Annals of Statistics*, vol. 43, no. 1, pp. 276–298, 2015.

[19] J. Hansen, S. Hissam, and G. Moreno, “Statistical-Based WCET Estimation and Validation,” *Wcet*, no. January, pp. 123–133, 2009.

[20] S. Coles, *An Introduction to Statistical Modeling of Extreme Values*. Springer Series in Statistics, London: Springer London, 2001.

[21] F. Reghenzani, G. Massari, and W. Fornaciari, “The Misconception of Exponential Tail Upper-Bounding in Probabilistic Real-Time,” *IEEE Embedded Systems Letters*, vol. PP, no. 801137, pp. 1–1, 2018.

[22] R. L. SMITH, “Maximum likelihood estimation in a class of nonregular cases,” *Biometrika*, vol. 72, no. 1, pp. 67–90, 1985.

[23] E. S. Martins and J. R. Stedinger, “Generalized maximum-likelihood generalized extreme-value quantile estimators for hydrologic data,” *Water Resources Research*, vol. 36, pp. 737–744, mar 2000.

[24] Y. Dezalay and B. Garth, “L-Moments : Analysis and Estimation of Distributions Using Linear Combinations of Order Statistics,” *Society*, vol. 52, no. 1, pp. 1243–1248, 2008.

[25] C. B. Bell, “Distribution-free statistical tests,” *Technometrics*, vol. 12, no. 4, pp. 929–929, 1970.

[26] F. Guet, L. Santinelli, and J. Morio, “On the Reliability of the Probabilistic Worst-Case Execution Time Estimates,” *European Congress on Embedded Real Time Software and Systems 2016 (ERTS’16)*, p. 10, 2016.

[27] L. Santinelli, F. Guet, and J. Morio, “Revising Measurement-Based Probabilistic Timing Analysis,” in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 199–208, IEEE, apr 2017.

[28] G. Lima, D. Dias, and E. Barros, “Extreme Value Theory for Estimating Task Execution Time Bounds: A Careful Look,” in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, vol. 2016-Augus, pp. 200–211, IEEE, jul 2016.

[29] S. Edgar and A. Burns, “Statistical analysis of WCET for scheduling,” in *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420)*, pp. 215–224, IEEE Comput. Soc, 2001.

[30] G. Bernat, A. Colin, and S. Petters, “pWCET: A tool for probabilistic worst-case execution time analysis of real-time systems,” *Report-University of York . . .* , pp. 1–18, 2003.

[31] A. Burns, G. Bernat, and I. Broster, “A Probabilistic Framework for Schedulability Analysis,” pp. 1–15, 2003.

[32] L. Kosmidis, E. Quinones, J. Abella, T. Vardanega, and F. J. Cazorla, “Achieving timing composability with measurement-based probabilistic timing analysis,” *16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2013*, 2014.

[33] A. Sklar, “Fonctions de R{é}partition {à} n Dimensions et Leurs Marges,” *Publications de L’Institut de Statistique de L’Universit{é} de Paris*, vol. 8, pp. 229–231, 1959.

[34] G. D. Makarov, “Estimates for the Distribution Function of a Sum of Two Random Variables When the Marginal Distributions are Fixed,” *Theory of Probability & Its Applications*, 1981.

[35] M. J. Frank, R. B. Nelsen, and B. Schweizer, “Best-possible bounds for the distribution of a sum - a problem of Kolmogorov,” *Probability Theory and Related Fields*, 1987.

[36] R. I. Davis and L. Cucu-Grosjean, “A Survey of Probabilistic Timing Analysis Techniques for Real-Time Systems,” *Leibniz Transactions on Embedded Systems (LITES)*, vol. 6, no. 1, pp. 3:1–3:60, 2019.

[37] S. Jimenez Gil, I. Bate, G. Lima, L. Santinelli, A. Gogonel, and L. Cucu-Grosjean, “Open Challenges for Probabilistic Measurement-Based Worst-Case Execution Time,” *IEEE Embedded Systems Letters*, vol. 9, pp. 69–72, sep 2017.

[38] L. Santinelli, J. Morio, G. Dufour, and D. Jacquemart, “On the Sustainability of the Extreme Value Theory for WCET Estimation,” *the 14th International Workshop on Worst-Case Execution Time (WCET) Analysis*, no. Wcet, pp. 21–30, 2014.

[39] F. Reghenzani, G. Massari, and W. Fornaciari, "chronovise: Measurement-Based Probabilistic Timing Analysis framework," *Journal of Open Source Software*, vol. 3, p. 711, aug 2018.

[40] S. Milutinovic, E. Mezzetti, J. Abella, T. Vardanega, and F. J. Cazorla, "On uses of extreme value theory fit for industrial-quality WCET analysis," in *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pp. 1–6, IEEE, jun 2017.

[41] F. J. Cazorla, J. Abella, J. Andersson, T. Vardanega, F. Vatrinet, I. Bate, I. Broster, M. Azkarate-Askasua, F. Wartel, L. Cucu, F. Cros, G. Farrall, A. Gogonel, A. Gianarro, B. Triquet, C. Hernandez, C. Lo, C. Maxim, D. Morales, E. Quinones, E. Mezzetti, L. Kosmidis, I. Aguirre, M. Fernandez, M. Slijepcevic, P. Conmy, and W. Talaboulma, "PROXIMA: Improving Measurement-Based Timing Analysis through Randomisation and Probabilistic Analysis," in *2016 Euromicro Conference on Digital System Design (DSD)*, pp. 276–285, IEEE, aug 2016.

[42] K. P. Silva, L. F. Arcaro, D. B. de Oliveira, and R. S. de Oliveira, "An Empirical Study on the Adequacy of MBPTA for Tasks Executed on a Complex Computer Architecture with Linux," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 321–328, IEEE, sep 2018.

[43] C. Curtsinger and E. D. Berger, "STABILIZER," in *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems - ASPLOS '13*, (New York, New York, USA), p. 219, ACM Press, 2013.

[44] L. Kosmidis, C. Curtsinger, E. Quinones, J. Abella, E. Berger, and F. J. Cazorla, "Probabilistic Timing Analysis on Conventional Cache Designs," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, (New Jersey), pp. 603–606, IEEE Conference Publications, 2013.

[45] F. Cros, L. Kosmidis, F. Wartel, D. Morales, J. Abella, I. Broster, and F. J. Cazorla, "Dynamic software randomisation: Lessons learned from an aerospace

case study," *Proceedings of the 2017 Design, Automation and Test in Europe, DATE 2017*, pp. 103–108, 2017.

[46] R. I. Davis, I. Bate, I. Broster, A. Burns, S. Hutchesson, and R.-r. Plc, "Transferring Real-Time Systems Research into Industrial Practice: Four Impact Case Studies," vol. 106, no. 7, pp. 7:1–7:24, 2018.

[47] A. Colin and G. Bernat, "Scope-tree: A program representation for symbolic worst-case execution time analysis," *Proceedings - Euromicro Conference on Real-Time Systems*, pp. 50–59, 2002.

[48] G. Bernat, R. Davis, N. Merriam, J. Tuffen, A. Gardner, M. Bennett, and D. Armstrong, "Identifying opportunities for worst-case execution time reduction in an avionics system," *Ada User Journal*, vol. 28, no. 3, pp. 189–194, 2007.

[49] R. S. Ltd., "RapiTime White Paper Worst-Case Execution Time Analysis," 2008.

[50] S. Law and I. Bate, "Achieving Appropriate Test Coverage for Reliable Measurement-Based Timing Analysis," *Proceedings - Euromicro Conference on Real-Time Systems*, vol. 2016-Augus, pp. 189–199, 2016.

[51] S. Law and I. Bate, "Achieving Appropriate Test Coverage for Reliable Measurement-Based Timing Analysis," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, vol. 2016-Augus, pp. 189–199, IEEE, jul 2016.

[52] B. Lesage, S. Law, and I. Bate, "TACO: An industrial case study of Test Automation for COverage," *International Conference on Real-Time Networks and Systems 2018 (RTNS'18)*, pp. 114–124, 2018.

[53] L. Kosmidis, J. Abella, F. Wartel, E. Quinones, A. Colin, and F. J. Cazorla, "PUB: Path Upper-Bounding for Measurement-Based Probabilistic Timing Analysis," in *2014 26th Euromicro Conference on Real-Time Systems*, pp. 276–287, IEEE, jul 2014.

[54] M. Ziccardi, E. Mezzetti, T. Vardanega, J. Abella, and F. J. Cazorla, "EPC: Extended Path Coverage for Measurement-Based Probabilistic Timing Analysis,"

in *2015 IEEE Real-Time Systems Symposium*, vol. 2016-Janua, pp. 338–349, IEEE, dec 2015.

[55] B. Lesage, D. Griffin, F. Soboczenski, I. Bate, and R. I. Davis, "A framework for the evaluation of measurement-based timing analyses," in *Proceedings of the 23rd International Conference on Real Time and Networks Systems - RTNS '15*, (New York, New York, USA), pp. 35–44, ACM Press, 2015.

[56] L. Palopoli, *Bringing Probabilistic Real – Time Guarantees to the Real World*. PhD thesis, 2018.

[57] C. B. Watkins and R. Walter, "Transitioning from federated avionics architectures to Integrated Modular Avionics," *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, pp. 1–10, 2007.

[58] P. J. Prisaznuk, "Arinc 653 role in Integrated Modular avionics (IMA)," *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, pp. 1–10, 2008.

[59] J. Rufino, S. Filipe, M. Coutinho, S. Santos, and J. Windsor, "ARINC 653 interface in RTEMS," *European Space Agency, (Special Publication) ESA SP*, no. SP-638, 2007.

[60] A. Betts and A. Marref, "WCET analysis of component-based systems using timing traces," *Proceedings - 2011 16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011*, pp. 13–22, 2011.

[61] A. Gaisler, "RCC User ' s Manual," no. February, pp. 1–21, 2018.

[62] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen WCET benchmarks: Past, present and future," *International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, 2010.

[63] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, pp. 3–30, jan 1998.

[64] C. Damman, G. Edison, F. Guet, E. Noulard, L. Santinelli, and J. Hugues, "Architectural performance analysis of FPGA synthesized LEON processors," in *Proceedings of the 27th International Symposium on Rapid System Prototyping*

Shortening the Path from Specification to Prototype - RSP '16, (New York, New York, USA), pp. 33–40, ACM Press, 2016.

[65] F. W. Scholz and M. A. Stephens, "K-sample Anderson–Darling tests," *Journal of the American Statistical Association*, vol. 82, no. 399, pp. 918–924, 1987.

[66] M. Garrido and J. Diebolt, "The {ET} test, a goodness-of-fit test for the distribution tail," *Methodology, Practice and Inference, second international conference on mathematical methods in reliability*, no. January 2001, pp. 427–430, 2000.

[67] N. 5001, "Nexus 5001 Debugging Interface Overview," 2004.

[68] "Arm development tools., http://www.arm.com," 2019.

[69] A. Betts, "Hybrid Measurement-Based WCET Analysis using Instrumentation Point Graphs," *Doctor*, no. February, 2010.

[70] A. Betts, N. Merriam, and G. Bernat, "Hybrid measurement-based WCET analysis at the source level using object-level traces," *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET)*, vol. 15, no. Wcet, pp. 54–63, 2010.

[71] Alexander Karlsson, "Real Time Trace Solution for LEON/GRLIB System-on-Chip," Master's thesis, Chalmers University of Technology, 2013.

[72] B. Dreyer, C. Hochberger, A. Lange, S. Wegener, and A. Weiss, "Continuous Non-Intrusive Hybrid WCET Estimation Using Waypoint Graphs," *Drops-Idn/6897*, vol. 55, no. 4, pp. 1–4, 2016.

[73] Altera Corporation, "What is an SoC FPGA?," *Altera Support Resources*, p. 4, 2014.

[74] Intel, "Avalon ® Interface Specifications," 2018.

[75] P. P. K N Vijeyakumar1, V Sumathy2 and S. Saravanakumar, "Design of High Speed Low Power Counter using Pipelining," *Journal of Scientific & Industrial Research,*, vol. 73, no. 4, pp. 1378–1382, 2014.

[76] J. Abella, C. Hernandez, E. Quinones, F. J. Cazorla, P. R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega, "WCET analysis methods:

Pitfalls and challenges on their trustworthiness," in *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pp. 1–10, IEEE, jun 2015.

[77] a. Colin and S. Petters, "Experimental evaluation of code properties for WCET analysis," *24th IEEE Real-Time Systems Symposium (RTSS 2003)*, pp. 190–199, 2003.

[78] S. M. Petters, "Execution-time profiles," *Technical Report, National ICT Australia*, pp. 1–6, 2007.

[79] P. Embrechts, "Copulas: A personal view," tech. rep.

[80] P. Arbenz, P. Embrechts, and G. Puccetti, "The GAEP algorithm for the fast computation of the distribution of a function of dependent random variables," *Stochastics*, vol. 84, no. 5-6, pp. 569–597, 2012.

[81] F. Miller, A. Vandome, and M. John, *Inverse Transform Sampling*. VDM Publishing, 2010.

[82] J. Dißmann, E. C. Brechmann, C. Czado, and D. Kurowicka, "Selecting and estimating regular vine copulae and application to financial returns," *Computational Statistics and Data Analysis*, vol. 59, no. 1, pp. 52–69, 2013.

[83] U. Schepsmeier, J. Stoeber, E. C. Brechmann, B. Graeler, T. Nagler, and T. Erhardt, *VineCopula: Statistical Inference of Vine Copulas*, 2018. R package version 2.1.8.

[84] R Core Team, *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2019.

[85] U. Schepsmeier, "A goodness-of-fit test for regular vine copula models," *Econometric Reviews*, pp. 1–22, jun 2013.

[86] A. Ghalanos, *spd: Semi-Parametric Distribution.*, 2015. R package version 2.0-1.

[87] ECSS-Secretariat, "Ecss-e-st-70-41c - telemetry and telecommand packet utilization," tech. rep., ESA-ESTEC Requirements and Standards Division, 2016.

[88] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, “Measurement-Based Timing Analysis,” pp. 430–444, 2008.

[89] S. Bünte, M. Zolda, and R. Kirner, “Let’s get less optimistic in measurement-based timing analysis,” *SIES 2011 - 6th IEEE International Symposium on Industrial Embedded Systems, Conference Proceedings*, pp. 204–212, 2011.

[90] S. Bünte, M. Zolda, M. Tautschnig, and R. Kirner, “Improving the confidence in measurement-based timing analysis,” *Proceedings - 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2011*, pp. 144–151, 2011.

[91] A. Trujillo-Ortiz, “Andarksamtest, matlab central file exchange,” 2007.

APPENDIX A

APPENDIX FOR CHAPTER 3

A.1 Independent and Identically Distributed Statistical Tests

Listing A.1: MATLAB Code for iidtest

```
function [] = iidtest(diff)

    diff = reshape(diff,[],1);

    sample_size_per_iter = floor(length(diff) * 0.1)

    for i = 1:sample_size_per_iter
        rng('shuffle');
        x1_ind = randi(length(diff) - sample_size_per_iter);
        x2_ind = randi(length(diff) - sample_size_per_iter);

        %Two-sample Kolmogorov-Smirnov test (identicality test)
        X1 = double(diff(x1_ind:x1_ind+sample_size_per_iter-1));
        X2 = double(diff(x2_ind:x2_ind+sample_size_per_iter-1));
        [h,p_ks(i)] = kstest2(X1,X2,'Alpha',0.05);

        %K-Sample Anderson-Darling Test (identicality test)
        X1 = double(diff(x1_ind:x1_ind+sample_size_per_iter-1));
        X2 = double(diff(x2_ind:x2_ind+sample_size_per_iter-1));
        X1(:,2) = 1;
        X2(:,2) = 2;
        p_ad(i) = AnDarksamtest([X1; X2],0.05);

        %LB Test (independency test)
        X1 = double(diff(x1_ind:x1_ind+sample_size_per_iter-1));
        res = X1 - mean(X1);
        [h,p_lb(i,:)] = lbqtest(res,'lags',[2,5,10,20,50]);

        %WW(runs) Test (independency test)
        X1 = double(diff(x1_ind:x1_ind+sample_size_per_iter-1));
        [h,p_ww(i)] = runstest(X1,median(X1));
    end

    boxplot([p_ks', p_ad', p_lb, p_ww'],
        'Labels',
        {'KS','AD','LB02', 'LB05', 'LB10', 'LB20', 'LB50', 'WW'})
    xlabel('Applicability statistical tests')
    ylabel('p-value distributions)')
    title('Statistical tests p-values')

end
```

`AnDarksamtest` function is fetched from [91].

A.2 Block Maxima for GEV

Listing A.2: MATLAB Code for getbm

```
function [diff_bm] = getbm(diff, bs)
% Example usage : diff_bm_fir = getbm(diff_fir, 50);
% Where diff_fir is the execution time array in clock count
% 50 corresponds to the block size

    diff = reshape(diff, [], 1);

    remainder = mod(length(diff), bs);

    diff_tmp = [diff; zeros(mod(bs - remainder, bs), 1)];

    diff_bm = reshape(diff_tmp, bs, []);
    diff_bm = max(diff_bm);
    diff_bm = double(diff_bm);

end
```

A.3 L-Moments Parameter Estimation for GEV

Listing A.3: MATLAB Code for lmomgev

```
function [paramEsts] = lmomgev(diff_bm)
% Example usage : paramEstsLmomFir = lmomgev(diff_bm_fir);
% Where diff_bm_fir is the block maxima of diff_fir

    [L] = lmom(diff_bm, 4);

    z = 2/(3 + L(3)/L(2)) - (log(2)/log(3));

    k = 7.8590*z+ 2.9554*(z^2);

    scale = L(2)*k/((1 - (2^(-k)))*gamma(1+k));

    M = L(1) + scale*(gamma(1+k) - 1) / k;

    k = -k;

    paramEsts = [k, scale, M];
end
```

The formula is taken from [24].

A.4 MLE Parameter Estimation for Gumbel

Listing A.4: MATLAB Code for mlegumbel

```
% gevfit function embedded in MATLAB is used to estimate the parameters
    with MLE
% Example usage : paramEstsMLFir = gevfit(diff_bm_fir);
% Where diff_bm_fir is the block maxima of diff_fir
```

A.5 CRPS for GEV and Gumbel

Listing A.5: MATLAB Code for crpsgev

```
function [crps_values, crps_convergence_index] = crpsgev(diff)
% Example Usage : [crps_values_fir_gev, crps_convergence_index_fir_gev] =
    crpsgev(diff_fir);
% Where diff_fir corresponds to the original observations

crps_convergence_index = -1;
crps_values= [];

lowerIndex = 100;
stepSize = 50;
crpsThreshold = 0.001;
crpsConvergenceRound = 5;

xgrid = linspace(floor(min(diff) - (0.1 * min(diff))), floor(max(diff) +
    (0.1 * max(diff))), 1000);

j = 1;
convergenceCounter = 0;
for i=lowerIndex:stepSize:length(diff)
    diff_bm_current = getbm(diff(1:i), 50);
    paramEstsCurrent = lmomgev(diff_bm_current);      %GEV with L-Moment
    cdfCurrent = gevcdf(double(xgrid),paramEstsCurrent(1),paramEstsCurrent
        (2), paramEstsCurrent(3));

    if(i > lowerIndex)
        crps_values(j) = sum((cdfCurrent - cdfPrev).^2);
        if((crps_convergence_index == -1) && (crps_values(j) <
            crpsThreshold))
            convergenceCounter = convergenceCounter + 1;
            if(convergenceCounter == crpsConvergenceRound)
                crps_convergence_index = j;
            end
        else
            convergenceCounter = 0;
        end

        j = j+1;
    end

    cdfPrev = cdfCurrent;
end
end
```

Listing A.6: MATLAB Code for crpsgumbel

```
function [crps_values, crps_convergence_index] = crpsgumbel(diff)
% Example Usage : [crps_values_fir_gumbel,
    crps_convergence_index_fir_gumbel] = crpsgumbel(diff_fir);
% Where diff_fir corresponds to the original observations

crps_convergence_index = -1;
crps_values= [];

lowerIndex = 100;
stepSize = 50;
crpsThreshold = 0.001;
crpsConvergenceRound = 5;

xgrid = linspace(floor(min(diff) - (0.1 * min(diff))), floor(max(diff) +
    (0.1 * max(diff))), 1000);

j = 1;
convergenceCounter = 0;
for i=lowerIndex:stepSize:length(diff)
    diff_bm_current = getbm(diff(1:i), 50);
    paramEstsCurrent = evfit(diff_bm_current);      %Gumbel
    cdfCurrent = gevcdf(double(xgrid),0,paramEstsCurrent(2),
        paramEstsCurrent(1));

    if(i > lowerIndex)
        crps_values(j) = sum((cdfCurrent - cdfPrev).^2);
        if((crps_convergence_index == -1) && (crps_values(j) <
            crpsThreshold))
            convergenceCounter = convergenceCounter + 1;
            if(convergenceCounter == crpsConvergenceRound)
                crps_convergence_index = j;
            end
        else
            convergenceCounter = 0;
        end

        j = j+1;
    end

    cdfPrev = cdfCurrent;
end
end
```

A.6 Estimation of pWCET Values from EVT Distributions

Listing A.7: MATLAB Code for pwcetEVT

```
pdgev_fir = makedist('GeneralizedExtremeValue','k',paramEstsLmomFirMs(1),'
    sigma',paramEstsLmomFirMs(2),'mu',paramEstsLmomFirMs(3));
pdgev_select = makedist('GeneralizedExtremeValue','k',paramEstsLmomSelectMs
    (1),'sigma',paramEstsLmomSelectMs(2),'mu',paramEstsLmomSelectMs(3));
pdgev_janne = makedist('GeneralizedExtremeValue','k',paramEstsLmomJanneMs
    (1),'sigma',paramEstsLmomJanneMs(2),'mu',paramEstsLmomJanneMs(3));

icdf(pdgev_fir, 0.9999)
icdf(pdgev_select, 0.9999)
icdf(pdgev_janne, 0.9999)
icdf(pdgev_fir, 0.999999)
icdf(pdgev_select, 0.999999)
icdf(pdgev_janne, 0.999999)
icdf(pdgev_fir, 0.999999999)
icdf(pdgev_select, 0.999999999)
icdf(pdgev_janne, 0.999999999)

pdgumb_fir = makedist('GeneralizedExtremeValue','k',0,'sigma',
    paramEstsMLFir(2),'mu',paramEstsMLFir(1));
pdgumb_select = makedist('GeneralizedExtremeValue','k',0,'sigma',
    paramEstsMLSelect(2),'mu',paramEstsMLSelect(1));
pdgumb_janne = makedist('GeneralizedExtremeValue','k',0,'sigma',
    paramEstsMLJanne(2),'mu',paramEstsMLJanne(1));

icdf(pdgumb_fir, 0.9999)
icdf(pdgumb_select, 0.9999)
icdf(pdgumb_janne, 0.9999)
icdf(pdgumb_fir, 0.999999)
icdf(pdgumb_select, 0.999999)
icdf(pdgumb_janne, 0.999999)
icdf(pdgumb_fir, 0.999999999)
icdf(pdgumb_select, 0.999999999)
icdf(pdgumb_janne, 0.999999999)
```

$paramEstsLmomFirMs$, $paramEstsLmomSelectMs$ and $paramEstsLmomJanneMs$ parameters are estimated by using $lmomgev$ function. However, instead of using $diff\_bm\_fir$, $diff\_bm\_select$ and $diff\_bm\_janne$, those values are converted into milliseconds representations by dividing each element to 200000 just to directly obtain the results in milliseconds format. It is also valid for $paramEstsMLFir$, $paramEstsMLSelect$ and $paramEstsMLJanne$.

APPENDIX B

APPENDIX FOR CHAPTER 4

B.1 Glitch Filter

Listing B.1: VHDL Code for Glitch Filter

```
-- glitchFilter.vhd
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity glitchFilter is
  generic (
    gpio_length : integer := 64
  );
  port (
    clock_sink_clk                    : in  std_logic
        := '0';             --            clock_sink.clk
    reset_sink_reset                  : in  std_logic
        := '0';             --            reset_sink.reset_n
    avalon_slave_address              : in  std_logic_vector(1 downto 0)
        := (others => '0'); --          avalon_slave.address
    avalon_slave_writedata            : in  std_logic_vector(31 downto 0)
        := (others => '0'); --                      .writedata
    avalon_slave_write                : in  std_logic
        := '0';             --                      .write
    avalon_slave_readdata             : out std_logic_vector(31 downto 0);
                            --                      .readdata
    avalon_slave_read                 : in  std_logic
        := '0';             --                      .read
    noisy_input                       : in  std_logic_vector(gpio_length -
        1 downto 0)   := (others => '0'); --          conduit_input.
        gpio_signal
    filtered_output             : out std_logic_vector(gpio_length - 1
        downto 0)   := (others => '0') --          conduit_output.gpio_signal
  );
end entity glitchFilter;

architecture rtl of glitchFilter is

   type sample_reg is array (1 to 3) of std_logic_vector(noisy_input'range)
       ;
   signal samples : sample_reg; -- shift register of sampled inputs

    signal state_change : std_ulogic; -- flag for a change in the input
        state

   signal filter_cycles : unsigned(15 downto 0) :=    (others => '0'); --
       Number of clock cycles to filter

   signal count : unsigned(filter_cycles'range) :=    (others => '0'); --
       timer count
   signal timer_done : std_ulogic; -- timer flag

   function or_reduce( v : std_logic_vector(noisy_input'range) ) return
       std_ulogic is
    variable result : std_ulogic := '0';
   begin
     for i in v'range loop
      result := result or v(i);
     end loop;

     return result;
   end function;

   signal     cntrl          : std_logic_vector(31 downto 0);
   signal     data           : std_logic_vector(31 downto 0);
   signal     status         : std_logic_vector(31 downto 0);
begin

  filter_cycles <= unsigned(cntrl(15 downto 0));

  -- MM-Slave Write Process
  process (clock_sink_clk, reset_sink_reset)
  begin
    if reset_sink_reset = '0' then
      cntrl    <=   (others => '0');
      data     <=   (others => '0');
      status   <=   (others => '0');
    elsif rising_edge(clock_sink_clk) then
      if (avalon_slave_write = '1') then
        case avalon_slave_address is
          when "00" =>  cntrl  <= avalon_slave_writedata;
          when "01" =>  data   <= avalon_slave_writedata;
          when others =>  null;
        end case;
      end if;
    end if;
  end process;

  -- MM-Slave Read Process
  process (clock_sink_clk, reset_sink_reset)
  begin
    if reset_sink_reset = '0' then
      avalon_slave_readdata   <=  (others => '0');
    elsif rising_edge(clock_sink_clk) then
      if(avalon_slave_read = '1') then
        case avalon_slave_address is
          when "00" =>  avalon_slave_readdata   <= cntrl;
          when "01" =>  avalon_slave_readdata   <= data;
          when "10" =>  avalon_slave_readdata   <= status;
          when others =>  avalon_slave_readdata   <= (others => 'X');
        end case;
      end if;
    end if;
  end process;

  -- Synchronize the noisy input and detect changes in state

  -- This would normally be an inappropriate way to synchronize an array
      but
  -- since the filter logic is waiting for all inputs to become stable, the
  -- usual issues with skewed inputs will not appear at the filtered output
      .
  sync: process(clock_sink_clk, reset_sink_reset) is
  begin
    if reset_sink_reset = '0' then
      samples <= (samples'range => (samples(1)'range => '0'));
    elsif rising_edge(clock_sink_clk) then
      samples <= noisy_input & samples(1 to 2);
    end if;
  end process;

  state_change <= '1' when or_reduce(to_x01(samples(3)) xor to_x01(samples
      (2))) = '1' else '0';
  -- Run count down continuously. Reset count whenever a state change
     occurs.
  -- If the count reaches 0 then the input has been stable for the
     requested
  -- length of time.
  timer: process(clock_sink_clk, reset_sink_reset, filter_cycles) is
  begin
    if reset_sink_reset = '0' then
      count <= filter_cycles;
    elsif rising_edge(clock_sink_clk) then
      if state_change = '1' then -- unstable, initialize timer
        count <= filter_cycles;
      else -- counting
        count <= count - 1;
      end if;
    end if;
  end process;

  timer_done <= '1' when count = (count'range => '0') else '0';

  -- Update the filtered output whenever the input has been stable for
     enough
  -- cycles.
  capture: process(clock_sink_clk, reset_sink_reset) is
  begin
    if reset_sink_reset = '0' then
      filtered_output <= (filtered_output'range => '0');
    elsif rising_edge(clock_sink_clk) then
      if timer_done = '1' then
        filtered_output <= samples(3);
      end if;
    end if;
  end process;

end architecture rtl; -- of traceCapture
```

B.2 Pipelined Counter

Listing B.2: VHDL Code for Pipelined Counter

```
-- pipelinedCounter.vhd
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity pipelinedCounter is
  generic (
    width_g : integer := 64;
    parts_g : integer := 4
  );
  port (
    avalon_slave_address          : in  std_logic_vector(1 downto 0)   := (
        others => '0'); --          avalon_slave.address
    avalon_slave_writedata        : in  std_logic_vector(31 downto 0)  := (
        others => '0'); --                      .writedata
    avalon_slave_write            : in  std_logic                      :=
        '0';            --                      .write
    avalon_slave_readdata         : out std_logic_vector(31 downto 0);
                           --                       .readdata
    avalon_slave_read             : in  std_logic                      :=
        '0';            --                      .read
    avalon_streaming_source_data  : out std_logic_vector(width_g - 1 downto
         0);                   -- avalon_streaming_source.data
    avalon_streaming_source_valid : out std_logic;
                                                      --                     .
        valid
    avalon_streaming_source_ready : in  std_logic                            :=
        '0';                --                    .ready
    clock_sink_clk                : in  std_logic                            :=
        '0';                --                 clock_sink.clk
    reset_sink_reset              : in  std_logic                            :=
        '0'                 --                 reset_sink.reset_n
  );
end entity pipelinedCounter;

architecture rtl of pipelinedCounter is
    constant  part_width_c    : natural := width_g / parts_g;
    signal    almost_tick_r   : std_logic_vector(parts_g - 1 downto 0);
    signal    count_r        : std_logic_vector(width_g - 1 downto 0);
   signal     running        : std_logic;
   signal     set_running     : std_logic;
   signal     clear_running   : std_logic;
   signal     enable         : std_logic;

   signal     cntrl          : std_logic_vector(31 downto 0);
   signal     data           : std_logic_vector(31 downto 0);
   signal     status         : std_logic_vector(31 downto 0);
begin

  avalon_streaming_source_data <= count_r;
  avalon_streaming_source_valid <= enable;

  set_running <= cntrl(31);
  clear_running <=  cntrl(31);
  enable <= running and avalon_streaming_source_ready;

  -- MM-Slave Write Process
  process (clock_sink_clk, reset_sink_reset)
  begin
    if reset_sink_reset = '0' then
      cntrl    <=   (others => '0');
      data     <=   (others => '0');
      status   <=   (others => '0');
    elsif rising_edge(clock_sink_clk) then
      if (avalon_slave_write = '1') then
        case avalon_slave_address is
          when "00" =>  cntrl  <= avalon_slave_writedata;
          when "01" =>  data   <= avalon_slave_writedata;
          when others =>  null;
        end case;
      end if;
    end if;
  end process;

  -- MM-Slave Read Process
  process (clock_sink_clk, reset_sink_reset)
  begin
    if reset_sink_reset = '0' then
      avalon_slave_readdata    <=   (others => '0');
    elsif rising_edge(clock_sink_clk) then
      if(avalon_slave_read = '1') then
        case avalon_slave_address is
          when "00" =>  avalon_slave_readdata   <= cntrl;
          when "01" =>  avalon_slave_readdata   <= data;
          when "10" =>  avalon_slave_readdata   <= status;
          when others =>  avalon_slave_readdata   <= (others => 'X');
        end case;
      end if;
    end if;
  end process;

  -- Running state process
  process (clock_sink_clk, reset_sink_reset)
  begin
    if reset_sink_reset = '0' then
      running    <=  '0';
    elsif rising_edge(clock_sink_clk) then
      if(set_running = '1') then
        running <=  '1';
      elsif(clear_running = '1') then
        running <=  '0';
      end if;
    end if;
  end process;

  -- Main process
  process (clock_sink_clk, reset_sink_reset)
    variable part_v:  unsigned(part_width_c - 1 downto 0);
    variable tick_v:  std_logic;
  begin
    if reset_sink_reset = '0' then
      count_r <= (others => '0');
      almost_tick_r <= (others => '0');
      --tick <= '0';
    elsif rising_edge(clock_sink_clk) then

      tick_v := enable;
      for i in 0 to parts_g - 1 loop
        part_v := unsigned(count_r((i + 1) * part_width_c - 1 downto i *
            part_width_c));

        if tick_v = '1' then
          -- Value is max - 1?
          if part_v = to_unsigned(2**part_width_c - 2, part_width_c) then
            almost_tick_r(i) <= '1';
          else
            almost_tick_r(i) <= '0';
          end if;

          part_v := part_v + 1;
        end if;

        count_r((i + 1) * part_width_c - 1 downto i * part_width_c) <=
            std_logic_vector(part_v);

        tick_v := tick_v and almost_tick_r(i);
      end loop;
      --tick <= tick_v;
    end if;
  end process;

end architecture rtl; -- of pipelinedCounter
```

B.3 Trace Capture

Listing B.3: VHDL Code for Trace Capture

```
-- traceCapture.vhd
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity traceCapture is
  generic (
    gpio_length : integer := 64;
    pipeline_counter_length : integer := 64
  );
  port (
    clock_sink_clk                    : in  std_logic
        := '0';               --            clock_sink.clk
    reset_sink_reset                  : in  std_logic
        := '0';               --            reset_sink.reset_n
    avalon_slave_address              : in  std_logic_vector(1 downto 0)
        := (others => '0'); --          avalon_slave.address
    avalon_slave_writedata            : in  std_logic_vector(31 downto 0)
        := (others => '0'); --                      .writedata
    avalon_slave_write                : in  std_logic
        := '0';               --                      .write
    avalon_slave_readdata              : out std_logic_vector(31 downto 0);
                                 --                          .readdata
    avalon_slave_read                  : in  std_logic
        := '0';                  --                          .read
    pipeline_counter_sink_data         : in  std_logic_vector(
        pipeline_counter_length - 1 downto 0) := (others => '0'); --
        pipeline_counter_sink.data
    pipeline_counter_sink_valid        : in  std_logic
        := '0';                  --                          .valid
    pipeline_counter_sink_ready        : out std_logic;
                                                   --                        .
        ready
    input_signal                       : in  std_logic_vector(gpio_length -
        1 downto 0)   := (others => '0'); --         conduit_input.
        gpio_signal
    trace_stream_source_data           : out std_logic_vector(127 downto 0);
                                 --   trace_stream_source.data
    trace_stream_source_valid          : out std_logic;
                                                   --                        .
        valid
    trace_stream_source_ready          : in  std_logic
        := '0';                  --                          .ready

    output_led                  : out std_logic_vector(2 downto 0) --debug
        purposes
  );
end entity traceCapture;

architecture rtl of traceCapture is

   TYPE State_type IS (IDLE, STREAM_VALID, ACK_VALID, ACK_INPUT_FALL);
   signal capture_state : State_Type;
   signal output_aligner    : std_logic_vector(63 - gpio_length downto 0);

   signal     running       : std_logic;
   signal     set_running     : std_logic;
   signal     clear_running   : std_logic;

   signal     current_count   : std_logic_vector(pipeline_counter_sink_data
       'RANGE);
   signal   count_sink_ready  : std_logic;
   signal   latch_counter    : std_logic;

   signal   source_valid     : std_logic;

   signal     cntrl          : std_logic_vector(31 downto 0);
   signal     data           : std_logic_vector(31 downto 0);
   signal     status         : std_logic_vector(31 downto 0);
begin

  pipeline_counter_sink_ready <= count_sink_ready;
  latch_counter <= running and pipeline_counter_sink_valid;

  set_running <= cntrl(31);
  clear_running <=  cntrl(31);

  trace_stream_source_valid <= source_valid;
  trace_stream_source_data <= current_count & input_signal & output_aligner
      ;
  ---trace_stream_source_sop <= source_valid;
  ---trace_stream_source_eop <= source_valid;

  output_led(0) <= source_valid;
  output_led(1) <= trace_stream_source_ready;
  output_led(2) <= (not input_signal(input_signal'HIGH)) and source_valid;

  -- MM-Slave Write Process
  process (clock_sink_clk, reset_sink_reset)
  begin
    if reset_sink_reset = '0' then
      cntrl    <=  (others => '0');
      data     <=  (others => '0');
      status   <=  (others => '0');
      output_aligner <= (others => '0');
    elsif rising_edge(clock_sink_clk) then
      if (avalon_slave_write = '1') then
        case avalon_slave_address is
          when "00" =>  cntrl <= avalon_slave_writedata;
          when "01" =>  data  <= avalon_slave_writedata;
          when others =>  null;
        end case;
      end if;
    end if;
  end process;

  -- MM-Slave Read Process
  process (clock_sink_clk, reset_sink_reset)
  begin
    if reset_sink_reset = '0' then
      avalon_slave_readdata   <=  (others => '0');
    elsif rising_edge(clock_sink_clk) then
      if(avalon_slave_read = '1') then
        case avalon_slave_address is
          when "00" =>  avalon_slave_readdata   <= cntrl;
          when "01" =>  avalon_slave_readdata   <= data;
          when "10" =>  avalon_slave_readdata   <= status;
          when others =>  avalon_slave_readdata   <= (others => 'X');
        end case;
      end if;
    end if;
  end process;

  -- Running state process
  process (clock_sink_clk, reset_sink_reset)
  begin
    if reset_sink_reset = '0' then
      running   <=  '0';
    elsif rising_edge(clock_sink_clk) then
      if(set_running = '1') then
        running <=  '1';
      elsif(clear_running = '1') then
        running <=  '0';
      end if;
    end if;
  end process;

  -- Pipeline Counter Latch Process
  process (clock_sink_clk, reset_sink_reset, pipeline_counter_sink_data)
  begin
    if reset_sink_reset = '0' then
      current_count     <=  (others => '0');
      count_sink_ready  <=    '0';
    elsif rising_edge(clock_sink_clk) then
      count_sink_ready  <=    '1';
      if(latch_counter = '1') then
        current_count   <=    pipeline_counter_sink_data;
      end if;
    end if;
  end process;

  --  Capture Process
  process (clock_sink_clk, reset_sink_reset, input_signal,
      trace_stream_source_ready)
    variable cntr:  UNSIGNED(1 downto 0);
  begin
    if reset_sink_reset = '0' then
      capture_state <= IDLE;
      source_valid <= '0';
      cntr := (others => '0');
    elsif rising_edge(clock_sink_clk) then
      if running = '1' then
        case capture_state is
          when IDLE =>
            if input_signal(input_signal'HIGH) = '1' and
                trace_stream_source_ready = '1' then
              capture_state   <=  STREAM_VALID;
            end if;
          when STREAM_VALID =>
            if input_signal(input_signal'HIGH) = '1' and
                trace_stream_source_ready = '1' then
              source_valid  <=  '1'; -- Danger
              capture_state <=  ACK_VALID;
            else
              capture_state <=  IDLE;
            end if;
          when ACK_VALID =>
            if source_valid = '1' then
              source_valid <= '0';
              capture_state <=  ACK_INPUT_FALL;
              cntr := (others => '0');
            else
              capture_state   <=  STREAM_VALID;
            end if;
          when ACK_INPUT_FALL =>
            if input_signal(input_signal'HIGH) = '0' then
              if cntr = 3 then
                capture_state <=  IDLE;
                cntr := (others => '0');
              else
                cntr := cntr + 1;
              end if;
            end if;
        end case;
      end if;
    end if;
  end process;

end architecture rtl; -- of traceCapture
```

APPENDIX C

APPENDIX FOR CHAPTER 5

C.1 Sequential Negatively Dependent Insertion Sort Program

Listing C.1: R Code for Neg. Dep. Example

```
library(R.matlab)
library(copula)
library(VineCopula)
library(psych)
library(extRemes)
library(ggplot2)
library(plotly)
library(spd)
library(PerformanceAnalytics)
library(pracma)
library(numbers)
library(data.table)
library(POT)

etp <- function(X)
{
  X_hist = as.data.frame(table(X))

  X_val <-as.numeric(as.character(X_hist[,1]))
  X_p <-as.numeric(as.character(X_hist[,2]))
  X_p <- X_p/sum(X_p)

  W = cbind(X_val, X_p)
  W
}

eecdf <- function(etpX)
{
  etpX[,2] = cumsum(etpX[,2])
  etpX[,2] = 1-etpX[,2]
  etpX
}

reduce_etp <- function(X, binCount)
{
  if(binCount == 0)
  {
    W = X
    W
  }
  else
  {
    px_p = X[,2]
    px_v = X[,1]

    stepSize = ceiling((length(px_p) - 1) / binCount)
    remainder = mod(length(px_p) - 1 ,stepSize)

    px_v_start = px_v[1]
    px_p_start = px_p[1]

    px_v_end = tail(px_v, 1)
    px_p_end = sum(tail(px_p, remainder))

    px_v = matrix(px_v[2:(length(px_v)-remainder)], nrow=stepSize)
    px_v = px_v[stepSize,]

    px_p = matrix(px_p[2:(length(px_p)-remainder)], nrow=stepSize)
    px_p = colSums(px_p)

    px_v = c(px_v_start, px_v)
    px_p = c(px_p_start, px_p)

    if(remainder > 0)
    {
      px_v = c(px_v, px_v_end)
      px_p = c(px_p, px_p_end)
    }

    px_p = px_p / sum(px_p)

    W = cbind(px_v, px_p)
    W
  }
}

indepConv <- function(etpX, etpY)
{
  Z_vals = outer(etpX[,1], t(etpY[,1]), FUN="+")
  Z_probs = etpX[,2] %*% t(etpY[,2])

  W = data.table(Vals = as.vector(Z_vals), Probs = as.vector(Z_probs), key=
      "Vals")

  W = W[, list(Probs=sum(Probs)), by=Vals]
  W = cbind(W$Vals, W$Probs)
  W
}

biasedConv <- function(etpX, etpY)
{
  X_val = etpX[,1]
  X_p = etpX[,2]

  Y_val = etpY[,1]
  Y_p = etpY[,2]

  i = length(X_p)
  j = length(Y_p)

  p_x = X_p[i]
  p_y = Y_p[j]
  val_x =  X_val[i]
  val_y =  Y_val[j]

  etpZ = c()

  while ((i > 0) || (j > 0))
  {
    p = min(p_x,p_y)

    etpZ = rbind(etpZ, c(val_x + val_y, p))

    p_x = p_x - p
    p_y = p_y - p

    if(i == 1 && j == 1)
    {
      i = 0;
      j = 0;
    }
    while((p_x <= 10^(-10)) && (i > 1))
    {
      i = i - 1;
      p_x = X_p[i]
      val_x =  X_val[i]
    }

    while((p_y <= 10^(-10)) && (j > 1))
    {
      j = j - 1;
      p_y = Y_p[j]
      val_y =  Y_val[j]
    }
  }

  etpZ = apply(etpZ,2,rev)

  etpZ

}

pwcetComonNegDep <- function(result, etpDownSampling)
{
  test_entry = etp(as.vector(result$test_entry))
  f1_endtoend = etp(as.vector(result$f1_ee))
  f1_ret = etp(as.vector(result$f1_ret))
  f2_endtoend = etp(as.vector(result$f2_ee))
  f2_ret = etp(as.vector(result$f2_ret))

  test_self_pwcet_etp = biasedConv(reduce_etp(test_entry, etpDownSampling),
      reduce_etp(f1_ret, etpDownSampling))
  test_self_pwcet_etp = biasedConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),reduce_etp(f2_ret, etpDownSampling))

  f1_pwcet_etp = f1_endtoend
  f2_pwcet_etp = f2_endtoend

  test_pwcet_etp = biasedConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),reduce_etp(f1_pwcet_etp, etpDownSampling))
  test_pwcet_etp = biasedConv(reduce_etp(test_pwcet_etp, etpDownSampling),
      reduce_etp(f2_pwcet_etp, etpDownSampling))
  test_pwcet_etp = reduce_etp(test_pwcet_etp, etpDownSampling)

  test_pwcet_etp
}

pwcetIndepNegDep <- function(result, etpDownSampling)
{
  test_entry = etp(as.vector(result$test_entry))
  f1_endtoend = etp(as.vector(result$f1_ee))
  f1_ret = etp(as.vector(result$f1_ret))
  f2_endtoend = etp(as.vector(result$f2_ee))
  f2_ret = etp(as.vector(result$f2_ret))

  test_self_pwcet_etp = indepConv(reduce_etp(test_entry, etpDownSampling),
      reduce_etp(f1_ret, etpDownSampling))
  test_self_pwcet_etp = indepConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),reduce_etp(f2_ret, etpDownSampling))

  f1_pwcet_etp = f1_endtoend
  f2_pwcet_etp = f2_endtoend

  test_pwcet_etp = indepConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),reduce_etp(f1_pwcet_etp, etpDownSampling))
  test_pwcet_etp = indepConv(reduce_etp(test_pwcet_etp, etpDownSampling),
      reduce_etp(f2_pwcet_etp, etpDownSampling))
  test_pwcet_etp = reduce_etp(test_pwcet_etp, etpDownSampling)

  test_pwcet_etp
}

data <- readMat("/Users/lvnt/Desktop/captures/program_reverse_sort.mat")

result <- convertData(data)
```

```
testEE = result$test_ee
testProgramEntry = result$test_entry
insertSortAsc = result$f1_ee
insertSortAscRet = result$f1_ret
insertSortDesc = result$f2_ee
insertSortDescRet = result$f2_ret

testProgramSelf = testProgramEntry + insertSortAscRet + insertSortDescRet
testProgramSub = insertSortAsc + insertSortDesc

df_testProgram <- data.frame(u = pobs(testProgramSelf), v = pobs(
    testProgramSub))
df_testProgramSelf <- data.frame(u = pobs(testProgramEntry), v = pobs(
    insertSortAscRet), k = pobs(insertSortDescRet))
df_testProgramSub <- data.frame(u = pobs(insertSortAsc), v = pobs(
    insertSortDesc))

chart.Correlation(df_testProgram, histogram=TRUE, pch=19)
chart.Correlation(df_testProgramSelf, histogram=TRUE, pch=19)
chart.Correlation(df_testProgramSub, histogram=TRUE, pch=19)

RVM_testProgram <- RVineStructureSelect(df_testProgram, c(1:6), indeptest =
     TRUE)
RVM_testProgramSelf <- RVineStructureSelect(df_testProgramSelf, c(1:6),
    indeptest = TRUE)
RVM_testProgramSub <- RVineStructureSelect(df_testProgramSub, c(1:6),
    indeptest = TRUE)

RVineGofTest(df_testProgram, RVM_testProgram, method="ECP2", statistic = "
    CvM")
RVineGofTest(df_testProgramSelf, RVM_testProgramSelf, method="ECP2",
    statistic = "CvM")
RVineGofTest(df_testProgramSub, RVM_testProgramSub, method="ECP2",
    statistic = "CvM")

simdata_testProgram <- RVineSim(1000000, RVM_testProgram)
simdata_testProgramSelf <- RVineSim(1000000, RVM_testProgramSelf)
simdata_testProgramSub <- RVineSim(1000000, RVM_testProgramSub)

tails_insertSortAsc <- spdfit(insertSortAsc, upper = 0.9, lower = 0.1)
tails_insertSortDesc <- spdfit(insertSortDesc, upper = 0.9, lower = 0.1)

testProgramEntry_cdf = etp(as.vector(testProgramEntry))
testProgramEntry_cdf[,2] = cumsum(testProgramEntry_cdf[,2])

insertSortAscRet_cdf = etp(as.vector(insertSortAscRet))
insertSortAscRet_cdf[,2] = cumsum(insertSortAscRet_cdf[,2])

insertSortDescRet_cdf = etp(as.vector(insertSortDescRet))
insertSortDescRet_cdf[,2] = cumsum(insertSortDescRet_cdf[,2])


testProgramEntry_real = findInterval(simdata_testProgramSelf[,1],
    testProgramEntry_cdf[,2])+1
testProgramEntry_real = testProgramEntry_cdf[,1][testProgramEntry_real]

insertSortAscRet_real = findInterval(simdata_testProgramSelf[,2],
    insertSortAscRet_cdf[,2])+1
insertSortAscRet_real = insertSortAscRet_cdf[,1][insertSortAscRet_real]

insertSortDescRet_real = findInterval(simdata_testProgramSelf[,3],
    insertSortDescRet_cdf[,2])+1
insertSortDescRet_real = insertSortDescRet_cdf[,1][insertSortDescRet_real]

testProgramSelf_W = testProgramEntry_real + insertSortAscRet_real +
    insertSortDescRet_real
testProgramSelf_cdf = etp(as.vector(testProgramSelf_W))
testProgramSelf_cdf[,2] = cumsum(testProgramSelf_cdf[,2])

testProgramSub_W = cbind(qspd(simdata_testProgramSub[,1], tails_
    insertSortAsc), qspd(simdata_testProgramSub[,2], tails_insertSortDesc))
testProgramSub_W = rowSums(testProgramSub_W)
testProgramSub_cdf = etp(as.vector(testProgramSub_W))
testProgramSub_cdf[,2] = cumsum(testProgramSub_cdf[,2])
```

```
testProgramSelf_real = findInterval(simdata_testProgram[,1],
    testProgramSelf_cdf[,2])+1
testProgramSelf_real = testProgramSelf_cdf[,1][testProgramSelf_real]

testProgramSub_real = findInterval(simdata_testProgram[,2], testProgramSub_
    cdf[,2])+1
testProgramSub_real = testProgramSub_cdf[,1][testProgramSub_real]

testProgram_W = testProgramSelf_real + testProgramSub_real


testProgram_ee_etp = etp(as.vector(testEE))
testProgram_spdCop_etp = etp(as.vector(testProgram_W))
testProgram_biasedConv_etp = pwcetComonNegDep(result, 0)
testProgram_independent_etp = pwcetIndepNegDep(result, 10000)

ee_x = testProgram_ee_etp[,1]
ee_y = eecdf(testProgram_ee_etp)[,2]

spdcop_x = testProgram_spdCop_etp[,1]
spdcop_y = eecdf(testProgram_spdCop_etp)[,2]

comon_x = testProgram_biasedConv_etp[,1]
comon_y = eecdf(testProgram_biasedConv_etp)[,2]

indep_x = testProgram_independent_etp[,1]
indep_y = eecdf(testProgram_independent_etp)[,2]

p <- plot_ly(x = ee_x, y = ee_y, type = 'scatter', mode = 'lines', name = '
    EE') %>%
      add_trace(x = spdcop_x, y = spdcop_y, type = 'scatter', mode = 'lines
          ', name = 'EVT-COP') %>%
      add_trace(x = comon_x, y = comon_y, type = 'scatter', mode = 'lines',
           name='RapiTime') %>%
      add_trace(x = indep_x, y = indep_y, type = 'scatter', mode = 'lines',
           name='Independent')
p
layout(p, yaxis = list(type = "log", exponentformat="power", showexponent="
    all"))

testProgram_ee_etp[,2] = cumsum(testProgram_ee_etp[,2])
testProgram_spdCop_etp[,2] = cumsum(testProgram_spdCop_etp[,2])
testProgram_biasedConv_etp[,2] = cumsum(testProgram_biasedConv_etp[,2])
testProgram_independent_etp[,2] = cumsum(testProgram_independent_etp[,2])

testProgram_spdCop_etp[,1][findInterval(0.99, testProgram_spdCop_etp[,2])
    +1] / 200000
testProgram_independent_etp[,1][findInterval(0.99, testProgram_independent_
    etp[,2])+1] / 200000
testProgram_biasedConv_etp[,1][findInterval(0.99, testProgram_biasedConv_
    etp[,2])+1] / 200000
testProgram_ee_etp[,1][findInterval(0.99, testProgram_ee_etp[,2])+1] /
    200000

testProgram_spdCop_etp[,1][findInterval(0.9999, testProgram_spdCop_etp[,2])
    +1] / 200000
testProgram_independent_etp[,1][findInterval(0.9999, testProgram_
    independent_etp[,2])+1] / 200000
testProgram_biasedConv_etp[,1][findInterval(0.9999, testProgram_biasedConv_
    etp[,2])+1] / 200000
testProgram_ee_etp[,1][findInterval(0.9999, testProgram_ee_etp[,2])+1] /
    200000

testProgram_spdCop_etp[,1][findInterval(0.999999, testProgram_spdCop_etp
    [,2])+1] / 200000
testProgram_independent_etp[,1][findInterval(0.999999, testProgram_
    independent_etp[,2])+1] / 200000
testProgram_biasedConv_etp[,1][findInterval(0.999999, testProgram_
    biasedConv_etp[,2])+1] / 200000
testProgram_ee_etp[,1][findInterval(0.999999, testProgram_ee_etp[,2])+1] /
    200000

testProgram_spdCop_etp[,1][findInterval(0.999999999, testProgram_spdCop_etp
    [,2])+1] / 200000
```

```
testProgram_independent_etp[,1][findInterval(0.999999999, testProgram_
    independent_etp[,2])+1] / 200000
testProgram_biasedConv_etp[,1][findInterval(0.999999999, testProgram_
    biasedConv_etp[,2])+1] / 200000
testProgram_ee_etp[,1][findInterval(0.999999999, testProgram_ee_etp[,2])+1]
     / 200000
```

C.2 Case-Study 1

Listing C.2: R Code for Case-Study 1

```
library(R.matlab)
library(copula)
library(VineCopula)
library(psych)
library(extRemes)
library(ggplot2)
library(plotly)
library(spd)
library(PerformanceAnalytics)
library(pracma)
library(numbers)
library(data.table)
library(POT)

etp <- function(X)
{
  X_hist = as.data.frame(table(X))

  X_val <-as.numeric(as.character(X_hist[,1]))
  X_p <-as.numeric(as.character(X_hist[,2]))
  X_p <- X_p/sum(X_p)

  W = cbind(X_val, X_p)
  W
}

reduce_etp <- function(X, binCount)
{
  if(binCount == 0)
  {
    W = X
    W
  }
  else
  {
    px_p = X[,2]
    px_v = X[,1]

    stepSize = ceiling((length(px_p) - 1) / binCount)

    remainder = mod(length(px_p) - 1 ,stepSize)

    px_v_start = px_v[1]
    px_p_start = px_p[1]

    px_v_end = tail(px_v, 1)
    px_p_end = sum(tail(px_p, remainder))

    px_v = matrix(px_v[2:(length(px_v)-remainder)], nrow=stepSize)
    px_v = px_v[stepSize,]

    px_p = matrix(px_p[2:(length(px_p)-remainder)], nrow=stepSize)
    px_p = colSums(px_p)

    px_v = c(px_v_start, px_v)
    px_p = c(px_p_start, px_p)
    if(remainder > 0)
    {
      px_v = c(px_v, px_v_end)
      px_p = c(px_p, px_p_end)
    }

    px_p = px_p / sum(px_p)

    W = cbind(px_v, px_p)
    W
  }
}

eecdf <- function(etpX)
{
  etpX[,2] = cumsum(etpX[,2])
  etpX[,2] = 1-etpX[,2]
  etpX
}

indepConv <- function(etpX, etpY)
{
  Z_vals = outer(etpX[,1], t(etpY[,1]), FUN="+")
  Z_probs = etpX[,2] %*% t(etpY[,2])

  W = data.table(Vals = as.vector(Z_vals), Probs = as.vector(Z_probs), key=
      "Vals")

  W = W[, list(Probs=sum(Probs)), by=Vals]
  W = cbind(W$Vals, W$Probs)
  W
}

biasedConv <- function(etpX, etpY)
{
  X_val = etpX[,1]
  X_p = etpX[,2]

  Y_val = etpY[,1]
  Y_p = etpY[,2]

  i = length(X_p)
  j = length(Y_p)

  p_x = X_p[i]
  p_y = Y_p[j]
  val_x =  X_val[i]
  val_y =  Y_val[j]

  etpZ = c()

  while ((i > 0) || (j > 0))
  {
    p = min(p_x,p_y)

    etpZ = rbind(etpZ, c(val_x + val_y, p))

    p_x = p_x - p
    p_y = p_y - p

    if(i == 1 && j == 1)
    {
      i = 0;
      j = 0;
    }

    while((p_x <= 10^(-10)) && (i > 1))
    {
      i = i - 1;
      p_x = X_p[i]
      val_x =  X_val[i]
    }

    while((p_y <= 10^(-10)) && (j > 1))
    {
      j = j - 1;
      p_y = Y_p[j]
      val_y =  Y_val[j]
    }
  }

  etpZ = apply(etpZ,2,rev)

  etpZ

}

pwcetComon <- function(result, etpDownSampling)
{
  test_entry = etp(as.vector(result$test_entry))
  f1_endtoend = etp(as.vector(result$f1_ee))
  f1_ret = etp(as.vector(result$f1_ret))
  f2_endtoend = etp(as.vector(result$f2_ee))
  f2_ret = etp(as.vector(result$f2_ret))
  f3_endtoend = etp(as.vector(result$f3_ee))
  f3_ret = etp(as.vector(result$f3_ret))
  f4_endtoend = etp(as.vector(result$f4_ee))
  f4_ret = etp(as.vector(result$f4_ret))
  f5_endtoend = etp(as.vector(result$f5_ee))
  f5_ret = etp(as.vector(result$f5_ret))
  f6_endtoend = etp(as.vector(result$f6_ee))
  f6_ret = etp(as.vector(result$f6_ret))
  f6_ret[,1] = 95 * f6_ret[,1]

  test_self_pwcet_etp = biasedConv(reduce_etp(test_entry, etpDownSampling),
      reduce_etp(f1_ret, etpDownSampling))
  test_self_pwcet_etp = biasedConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),reduce_etp(f2_ret, etpDownSampling))
  test_self_pwcet_etp = biasedConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),reduce_etp(f3_ret, etpDownSampling))
  test_self_pwcet_etp = biasedConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),reduce_etp(f5_ret, etpDownSampling))
  test_self_pwcet_etp = biasedConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),reduce_etp(f6_ret, etpDownSampling))

  f1_pwcet_etp = f1_endtoend
  f2_pwcet_etp = f2_endtoend
  f3_pwcet_etp = f3_endtoend
  f4_pwcet_etp = f4_endtoend
  f5_pwcet_etp = f5_endtoend
  f6_pwcet_etp = f6_endtoend

  f6_pwcet_etp_in_testProgram = f6_pwcet_etp
  f6_pwcet_etp_in_testProgram[,1] = 95 * f6_pwcet_etp_in_testProgram[,1]
  test_pwcet_etp = biasedConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),reduce_etp(f1_pwcet_etp, etpDownSampling))
  test_pwcet_etp = biasedConv(reduce_etp(test_pwcet_etp, etpDownSampling),
      reduce_etp(f2_pwcet_etp, etpDownSampling))
  test_pwcet_etp = biasedConv(reduce_etp(test_pwcet_etp, etpDownSampling),
      reduce_etp(f3_pwcet_etp, etpDownSampling))
  test_pwcet_etp = biasedConv(reduce_etp(test_pwcet_etp, etpDownSampling),
      reduce_etp(f5_pwcet_etp, etpDownSampling))
  test_pwcet_etp = biasedConv(reduce_etp(test_pwcet_etp, etpDownSampling),
      reduce_etp(f6_pwcet_etp_in_testProgram, etpDownSampling))
  test_pwcet_etp = reduce_etp(test_pwcet_etp, etpDownSampling)

  test_pwcet_etp
}

pwcetIndep <- function(result, etpDownSampling)
{
  test_entry = etp(as.vector(result$test_entry))
  f1_endtoend = etp(as.vector(result$f1_ee))
  f1_ret = etp(as.vector(result$f1_ret))
  f2_endtoend = etp(as.vector(result$f2_ee))
  f2_ret = etp(as.vector(result$f2_ret))
  f3_endtoend = etp(as.vector(result$f3_ee))
  f3_ret = etp(as.vector(result$f3_ret))
  f4_endtoend = etp(as.vector(result$f4_ee))
  f4_ret = etp(as.vector(result$f4_ret))
  f5_endtoend = etp(as.vector(result$f5_ee))
  f5_ret = etp(as.vector(result$f5_ret))
  f6_endtoend = etp(as.vector(result$f6_ee))
  f6_ret = etp(as.vector(result$f6_ret))

  test_self_pwcet_etp = indepConv(reduce_etp(test_entry, etpDownSampling),
      reduce_etp(f1_ret, etpDownSampling))
  test_self_pwcet_etp = indepConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),reduce_etp(f2_ret, etpDownSampling))
  test_self_pwcet_etp = indepConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),reduce_etp(f3_ret, etpDownSampling))
  test_self_pwcet_etp = indepConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),reduce_etp(f5_ret, etpDownSampling))

  f6_ret_pwcet_etp_in_testProgram = f6_ret
  for (i in 2:95)
  {
    f6_ret_pwcet_etp_in_testProgram = indepConv(reduce_etp(f6_ret_pwcet_etp
        _in_testProgram, etpDownSampling),
                                                  reduce_etp(f6_ret,
                                                      etpDownSampling))
  }


  test_self_pwcet_etp = indepConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),
                                  reduce_etp(f6_ret_pwcet_etp_in_
                                      testProgram, etpDownSampling))

  f1_pwcet_etp = f1_endtoend
  f2_pwcet_etp = f2_endtoend
  f3_pwcet_etp = f3_endtoend
  f4_pwcet_etp = f4_endtoend
  f5_pwcet_etp = f5_endtoend
  f6_pwcet_etp = f6_endtoend

  f6_pwcet_etp_in_testProgram = f6_pwcet_etp
  for (i in 2:95)
  {
    f6_pwcet_etp_in_testProgram = indepConv(reduce_etp(f6_pwcet_etp_in_
        testProgram, etpDownSampling),
                                            reduce_etp(f6_pwcet_etp,
                                                etpDownSampling))
  }
  test_pwcet_etp = indepConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),reduce_etp(f1_pwcet_etp, etpDownSampling))
  test_pwcet_etp = indepConv(reduce_etp(test_pwcet_etp, etpDownSampling),
      reduce_etp(f2_pwcet_etp, etpDownSampling))
  test_pwcet_etp = indepConv(reduce_etp(test_pwcet_etp, etpDownSampling),
      reduce_etp(f3_pwcet_etp, etpDownSampling))
  test_pwcet_etp = indepConv(reduce_etp(test_pwcet_etp, etpDownSampling),
      reduce_etp(f5_pwcet_etp, etpDownSampling))
  test_pwcet_etp = indepConv(reduce_etp(test_pwcet_etp, etpDownSampling),
      reduce_etp(f6_pwcet_etp_in_testProgram, etpDownSampling))
  test_pwcet_etp = reduce_etp(test_pwcet_etp, etpDownSampling)

  test_pwcet_etp
}

data <- readMat("/Users/lvnt/Desktop/captures/program_6func_deploop_indep_
    cond_scope.mat")

result <- convertData(data)

testEE = result$test_ee
testProgramEntry = result$test_entry
f1 = result$f1_ee
f1Ret = result$f1_ret
f2 = result$f2_ee
f2Ret = result$f2_ret
f3 = result$f3_ee
f3Ret = result$f3_ret
f4 = result$f4_ee
f4Ret = result$f4_ret
```

```
f5 = result$f5_ee
f5Ret = result$f5_ret
f6 = result$f6_ee
f6Ret = result$f6_ret

cond_sub = rbind(f4, f5)
cond_sub = colSums(cond_sub)
cond_self = rbind(f4Ret, f5Ret)
cond_self = colSums(cond_self)

loop_sub = result$f6_ee_cum
loop_cnt = result$f6_loop_cnt
loop_self = result$f6_ret_cum

testProgramSelf = testProgramEntry + f1Ret + f2Ret + f3Ret + cond_self +
    loop_self
testProgramSub = f1 + f2 + f3 + cond_sub + loop_sub

df_testProgram <- data.frame(u = pobs(testProgramSelf), v = pobs(
    testProgramSub))
df_testProgramSelf <- data.frame(u = pobs(testProgramEntry), v = pobs(f1Ret
    ), k = pobs(f2Ret), m = pobs(f3Ret), n = pobs(cond_self), p = pobs(loop
    _self))
df_testProgramSub <- data.frame(u = pobs(f1), v = pobs(f2), k = pobs(f3), m
     = pobs(cond_sub), n = pobs(loop_sub))

chart.Correlation(df_testProgram, histogram=TRUE, pch=19)
chart.Correlation(df_testProgramSelf, histogram=TRUE, pch=19)
chart.Correlation(df_testProgramSub, histogram=TRUE, pch=19)

RVM_testProgram <- RVineStructureSelect(df_testProgram, c(1:6), indeptest =
     TRUE)
RVM_testProgramSelf <- RVineStructureSelect(df_testProgramSelf, c(1:6),
    indeptest = TRUE)
RVM_testProgramSub <- RVineStructureSelect(df_testProgramSub, c(1:6),
    indeptest = TRUE)

RVineGofTest(df_testProgram, RVM_testProgram, method="ECP2", statistic = "
    CvM")
RVineGofTest(df_testProgramSelf, RVM_testProgramSelf, method="ECP2",
    statistic = "CvM")
RVineGofTest(df_testProgramSub, RVM_testProgramSub, method="ECP2",
    statistic = "CvM")

simdata_testProgram <- RVineSim(1000000, RVM_testProgram)
simdata_testProgramSelf <- RVineSim(1000000, RVM_testProgramSelf)
simdata_testProgramSub <- RVineSim(1000000, RVM_testProgramSub)

f4 = f4[ which(!f4 == 0)]
f5 = f5[ which(!f5 == 0)]

tails_f1 <- spdfit(f1, upper = 0.9, lower = 0.1)
tails_f2 <- spdfit(f2, upper = 0.9, lower = 0.1)
tails_f3 <- spdfit(f3, upper = 0.9, lower = 0.1)
tails_f4 <- spdfit(f4, upper = 0.9, lower = 0.1)
tails_f5 <- spdfit(f5, upper = 0.9, lower = 0.1)
tails_f6 <- spdfit(f6, upper = 0.9, lower = 0.1)

f4Ret = f4Ret[ which(!f4Ret == 0)]
f5Ret = f5Ret[ which(!f5Ret == 0)]
testProgramEntry_cdf = etp(as.vector(testProgramEntry))
testProgramEntry_cdf[,2] = cumsum(testProgramEntry_cdf[,2])

f1Ret_cdf = etp(as.vector(f1Ret))
f1Ret_cdf[,2] = cumsum(f1Ret_cdf[,2])

f2Ret_cdf = etp(as.vector(f2Ret))
f2Ret_cdf[,2] = cumsum(f2Ret_cdf[,2])

f3Ret_cdf = etp(as.vector(f3Ret))
f3Ret_cdf[,2] = cumsum(f3Ret_cdf[,2])

f4Ret_cdf = etp(as.vector(f4Ret))
f4Ret_cdf[,2] = cumsum(f4Ret_cdf[,2])
```

```
f5Ret_cdf = etp(as.vector(f5Ret))
f5Ret_cdf[,2] = cumsum(f5Ret_cdf[,2])

f6Ret_etp = etp(as.vector(f6Ret))
f6Ret_etp = reduce_etp(f6Ret_etp, 1000) #do not call this when biased conv.
     is applied
loop_self_cdf = f6Ret_etp

for (i in 2:max(loop_cnt))
{
  loop_self_cdf = reduce_etp(loop_self_cdf, 1000)
  loop_self_cdf = indepConv(loop_self_cdf,f6Ret_etp)
}

loop_self_cdf[,2] = cumsum(loop_self_cdf[,2])

testProgramEntry_real = findInterval(simdata_testProgramSelf[,1],
    testProgramEntry_cdf[,2])+1
testProgramEntry_real = testProgramEntry_cdf[,1][testProgramEntry_real]

f1Ret_real = findInterval(simdata_testProgramSelf[,2], f1Ret_cdf[,2])+1
f1Ret_real = f1Ret_cdf[,1][f1Ret_real]

f2Ret_real = findInterval(simdata_testProgramSelf[,3], f2Ret_cdf[,2])+1
f2Ret_real = f2Ret_cdf[,1][f2Ret_real]

f3Ret_real = findInterval(simdata_testProgramSelf[,4], f3Ret_cdf[,2])+1
f3Ret_real = f3Ret_cdf[,1][f3Ret_real]

f4Ret_real = findInterval(simdata_testProgramSelf[,5], f4Ret_cdf[,2])+1
f4Ret_real = f4Ret_cdf[,1][f4Ret_real]

f5Ret_real = findInterval(simdata_testProgramSelf[,5], f5Ret_cdf[,2])+1
f5Ret_real = f5Ret_cdf[,1][f5Ret_real]

cond_self_W = apply(cbind(f4Ret_real, f5Ret_real), 1, max)

loop_self_W = findInterval(simdata_testProgramSelf[,6], loop_self_cdf[,2])
    +1
loop_self_W = loop_self_cdf[,1][loop_self_W]

testProgramSelf_W = testProgramEntry_real + f1Ret_real + f2Ret_real + f3Ret
    _real + cond_self_W + loop_self_W
testProgramSelf_cdf = etp(as.vector(testProgramSelf_W))
testProgramSelf_cdf[,2] = cumsum(testProgramSelf_cdf[,2])


loop1_idx = c()
for (i in 1:length(loop_cnt))
{
  loop1_idx = c(loop1_idx,1:loop_cnt[i])
}
loop1_idx_pobs = pobs(loop1_idx)
f6_pobs = pobs(f6)

df_loop <- data.frame(u = loop1_idx_pobs, v = f6_pobs)
chart.Correlation(df_loop, histogram=TRUE, pch=19)
BiCopLoop1 <- BiCopSelect(loop1_idx_pobs, f6_pobs, familyset =c(1:6),
    indeptest = TRUE)

f6_etp = etp(as.vector(f6))
f6_etp = reduce_etp(f6_etp, 1000) #do not call this when biased conv. is
    applied
loop_sub_cdf = f6_etp

for (i in 2:max(loop_cnt))
{
  loop_sub_cdf = reduce_etp(loop_sub_cdf, 1000)
  loop_sub_cdf = indepConv(loop_sub_cdf,f6_etp)
}

loop_sub_cdf[,2] = cumsum(loop_sub_cdf[,2])

loop_sub_W = findInterval(simdata_testProgramSub[,5], loop_sub_cdf[,2])+1
loop_sub_W = loop_sub_cdf[,1][loop_sub_W]
```

```
cond_sub_W = apply(cbind(qspd( simdata_testProgramSub[,4], tails_f4), qspd(
     simdata_testProgramSub[,4], tails_f5)), 1, max)

f1_real = qspd( simdata_testProgramSub[,1], tails_f1)
f2_real = qspd( simdata_testProgramSub[,2], tails_f2)
f3_real = qspd( simdata_testProgramSub[,3], tails_f3)

testProgramSub_W = f1_real + f2_real + f3_real + cond_sub_W + loop_sub_W
testProgramSub_cdf = etp(as.vector(testProgramSub_W))
testProgramSub_cdf[,2] = cumsum(testProgramSub_cdf[,2])

testProgramSelf_real = findInterval(simdata_testProgram[,1],
    testProgramSelf_cdf[,2])+1
testProgramSelf_real = testProgramSelf_cdf[,1][testProgramSelf_real]

testProgramSub_real = findInterval(simdata_testProgram[,2], testProgramSub_
    cdf[,2])+1
testProgramSub_real = testProgramSub_cdf[,1][testProgramSub_real]

testProgram_W = testProgramSelf_real + testProgramSub_real


testProgram_ee_etp = etp(as.vector(testEE))
testProgram_spdCop_etp = etp(as.vector(testProgram_W))
testProgram_biasedConv_etp = pwcetComon(result, 0)
testProgram_independent_etp = pwcetIndep(result, 10000)

ee_x = testProgram_ee_etp[,1]
ee_y = eecdf(testProgram_ee_etp)[,2]

spdcop_x = testProgram_spdCop_etp[,1]
spdcop_y = eecdf(testProgram_spdCop_etp)[,2]

comon_x = testProgram_biasedConv_etp[,1]
comon_y = eecdf(testProgram_biasedConv_etp)[,2]

indep_x = testProgram_independent_etp[,1]
indep_y = eecdf(testProgram_independent_etp)[,2]

p <- plot_ly(x = ee_x, y = ee_y, type = 'scatter', mode = 'lines', name = '
    EE') %>%
      add_trace(x = spdcop_x, y = spdcop_y, type = 'scatter', mode = 'lines
          ', name = 'EVT-COP') %>%
      add_trace(x = comon_x, y = comon_y, type = 'scatter', mode = 'lines',
           name='RapiTime') %>%
      add_trace(x = indep_x, y = indep_y, type = 'scatter', mode = 'lines',
           name='Independent')
p
layout(p, yaxis = list(type = "log", exponentformat="power", showexponent="
    all"))

testProgram_ee_etp[,2] = cumsum(testProgram_ee_etp[,2])
testProgram_spdCop_etp[,2] = cumsum(testProgram_spdCop_etp[,2])
testProgram_biasedConv_etp[,2] = cumsum(testProgram_biasedConv_etp[,2])
testProgram_independent_etp[,2] = cumsum(testProgram_independent_etp[,2])

testProgram_spdCop_etp[,1][findInterval(0.99, testProgram_spdCop_etp[,2])
    +1] / 200000
testProgram_independent_etp[,1][findInterval(0.99, testProgram_independent_
    etp[,2])+1] / 200000
testProgram_biasedConv_etp[,1][findInterval(0.99, testProgram_biasedConv_
    etp[,2])+1] / 200000
testProgram_ee_etp[,1][findInterval(0.99, testProgram_ee_etp[,2])+1] /
    200000

testProgram_spdCop_etp[,1][findInterval(0.9999, testProgram_spdCop_etp[,2])
    +1] / 200000
testProgram_independent_etp[,1][findInterval(0.9999, testProgram_
    independent_etp[,2])+1] / 200000
testProgram_biasedConv_etp[,1][findInterval(0.9999, testProgram_biasedConv_
    etp[,2])+1] / 200000
testProgram_ee_etp[,1][findInterval(0.9999, testProgram_ee_etp[,2])+1] /
    200000
```

```
testProgram_spdCop_etp[,1][findInterval(0.999999, testProgram_spdCop_etp
    [,2])+1] / 200000
testProgram_independent_etp[,1][findInterval(0.999999, testProgram_
    independent_etp[,2])+1] / 200000
testProgram_biasedConv_etp[,1][findInterval(0.999999, testProgram_
    biasedConv_etp[,2])+1] / 200000
testProgram_ee_etp[,1][findInterval(0.999999, testProgram_ee_etp[,2])+1] /
    200000

testProgram_spdCop_etp[,1][findInterval(0.999999999, testProgram_spdCop_etp
    [,2])+1] / 200000
testProgram_independent_etp[,1][findInterval(0.999999999, testProgram_
    independent_etp[,2])+1] / 200000
testProgram_biasedConv_etp[,1][findInterval(0.999999999, testProgram_
    biasedConv_etp[,2])+1] / 200000
testProgram_ee_etp[,1][findInterval(0.999999999, testProgram_ee_etp[,2])+1]
     / 200000
```

C.3 Case-Study 2

Listing C.3: R Code for Case-Study 2

```
library(R.matlab)
library(copula)
library(VineCopula)
library(psych)
library(extRemes)
library(ggplot2)
library(plotly)
library(spd)
library(PerformanceAnalytics)
library(pracma)
library(numbers)
library(data.table)
library(POT)

eecdf <- function(etpX)
{
  etpX[,2] = cumsum(etpX[,2])
  etpX[,2] = 1-etpX[,2]
  etpX
}

etp <- function(X)
{
  X_hist = as.data.frame(table(X))

  X_val <-as.numeric(as.character(X_hist[,1]))
  X_p <-as.numeric(as.character(X_hist[,2]))
  X_p <- X_p/sum(X_p)

  W = cbind(X_val, X_p)
  W
}

reduce_etp <- function(X, binCount)
{
  if(binCount == 0)
  {
    W = X
    W
  }
  else
  {
    px_p = X[,2]
    px_v = X[,1]
    stepSize = ceiling((length(px_p) - 1) / binCount)

    remainder = mod(length(px_p) - 1 ,stepSize)

    px_v_start = px_v[1]
    px_p_start = px_p[1]

    px_v_end = tail(px_v, 1)
    px_p_end = sum(tail(px_p, remainder))

    px_v = matrix(px_v[2:(length(px_v)-remainder)], nrow=stepSize)
    px_v = px_v[stepSize,]

    px_p = matrix(px_p[2:(length(px_p)-remainder)], nrow=stepSize)
    px_p = colSums(px_p)

    px_v = c(px_v_start, px_v)
    px_p = c(px_p_start, px_p)

    if(remainder > 0)
    {
      px_v = c(px_v, px_v_end)
      px_p = c(px_p, px_p_end)
    }

    px_p = px_p / sum(px_p)

    W = cbind(px_v, px_p)
    W
  }
}

indepConv <- function(etpX, etpY)
{
  Z_vals = outer(etpX[,1], t(etpY[,1]), FUN="+")
  Z_probs = etpX[,2] %*% t(etpY[,2])

  W = data.table(Vals = as.vector(Z_vals), Probs = as.vector(Z_probs), key=
      "Vals")

  W = W[, list(Probs=sum(Probs)), by=Vals]
  W = cbind(W$Vals, W$Probs)
  W
}

biasedConv <- function(etpX, etpY)
{
  X_val = etpX[,1]
  X_p = etpX[,2]

  Y_val = etpY[,1]
  Y_p = etpY[,2]

  i = length(X_p)
  j = length(Y_p)

  p_x = X_p[i]
  p_y = Y_p[j]
  val_x =  X_val[i]
  val_y =  Y_val[j]

  etpZ = c()

  while ((i > 0) || (j > 0))
  {
    p = min(p_x,p_y)

    etpZ = rbind(etpZ, c(val_x + val_y, p))

    p_x = p_x - p
    p_y = p_y - p

    if(i == 1 && j == 1)
    {
      i = 0;
      j = 0;
    }

    while((p_x <= 10^(-10)) && (i > 1))
    {
      i = i - 1;
      p_x = X_p[i]
      val_x =  X_val[i]
    }

    while((p_y <= 10^(-10)) && (j > 1))
    {
      j = j - 1;
      p_y = Y_p[j]
      val_y =  Y_val[j]
    }
  }

  etpZ = apply(etpZ,2,rev)

  etpZ

}


pwcetComonCase2 <- function(result, etpDownSampling)
{
  f1 = result$f1_ee
  f1Ret = result$f1_ret
  f2 = result$f2_ee
  f2Ret = result$f2_ret
  f3 = result$f3_ee
  f3Ret = result$f3_ret
  f4 = result$f4_ee
  f4Ret = result$f4_ret
  f5 = result$f5_ee
  f5Ret = result$f5_ret
  f6 = result$f6_ee
  f6Ret = result$f6_ret
  f7 = result$f7_ee
  f7Ret = result$f7_ret

  f1 = f1[ which(!f1 == 0)]
  f1Ret = f1Ret[ which(!f1Ret == 0)]
  f2 = f2[ which(!f2 == 0)]
  f2Ret = f2Ret[ which(!f2Ret == 0)]
  f3 = f3[ which(!f3 == 0)]
  f3Ret = f3Ret[ which(!f3Ret == 0)]
  f4 = f4[ which(!f4 == 0)]
  f4Ret = f4Ret[ which(!f4Ret == 0)]
  f5 = f5[ which(!f5 == 0)]
  f5Ret = f5Ret[ which(!f5Ret == 0)]
  f6 = f6[ which(!f6 == 0)]
  f6Ret = f6Ret[ which(!f6Ret == 0)]
  f7 = f7[ which(!f7 == 0)]
  f7Ret = f7Ret[ which(!f7Ret == 0)]

  test_entry = etp(as.vector(result$test_entry))
  f1_endtoend = etp(as.vector(f1))
  f1_ret = etp(as.vector(f1Ret))
  f2_endtoend = etp(as.vector(f2))
  f2_ret = etp(as.vector(f2Ret))
  f3_endtoend = etp(as.vector(f3))
  f3_ret = etp(as.vector(f3Ret))
  f4_endtoend = etp(as.vector(f4))
  f4_ret = etp(as.vector(f4Ret))
  f6_endtoend = etp(as.vector(f6))
  f6_ret = etp(as.vector(f6Ret))
  f7_endtoend = etp(as.vector(f7))
  f7_ret = etp(as.vector(f7Ret))

  f3_ret[,1] = 5 * f3_ret[,1]
  f4_ret[,1] = 5 * f4_ret[,1]
  test_self_pwcet_etp = biasedConv(reduce_etp(test_entry, etpDownSampling),
      reduce_etp(f1_ret, etpDownSampling))
  test_self_pwcet_etp = biasedConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),reduce_etp(f2_ret, etpDownSampling))
  test_self_pwcet_etp = biasedConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),reduce_etp(f3_ret, etpDownSampling))
  test_self_pwcet_etp = biasedConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),reduce_etp(f4_ret, etpDownSampling))
  test_self_pwcet_etp = biasedConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),reduce_etp(f6_ret, etpDownSampling))

  f1_pwcet_etp = f1_endtoend
  f2_pwcet_etp = f2_endtoend
  f3_pwcet_etp = f3_endtoend
  f4_pwcet_etp = f4_endtoend
  f6_pwcet_etp = f6_endtoend

  f3_pwcet_etp_in_testProgram = f3_pwcet_etp
  f3_pwcet_etp_in_testProgram[,1] = 5 * f3_pwcet_etp_in_testProgram[,1]
  f4_pwcet_etp_in_testProgram = f4_pwcet_etp
  f4_pwcet_etp_in_testProgram[,1] = 5 * f4_pwcet_etp_in_testProgram[,1]
  test_pwcet_etp = biasedConv(reduce_etp(test_self_pwcet_etp,
      etpDownSampling),reduce_etp(f1_pwcet_etp, etpDownSampling))
  test_pwcet_etp = biasedConv(reduce_etp(test_pwcet_etp, etpDownSampling),
      reduce_etp(f2_pwcet_etp, etpDownSampling))
  test_pwcet_etp = biasedConv(reduce_etp(test_pwcet_etp, etpDownSampling),
      reduce_etp(f3_pwcet_etp_in_testProgram, etpDownSampling))
  test_pwcet_etp = biasedConv(reduce_etp(test_pwcet_etp, etpDownSampling),
      reduce_etp(f4_pwcet_etp_in_testProgram, etpDownSampling))
  test_pwcet_etp = biasedConv(reduce_etp(test_pwcet_etp, etpDownSampling),
      reduce_etp(f6_pwcet_etp, etpDownSampling))
  test_pwcet_etp = reduce_etp(test_pwcet_etp, etpDownSampling)

  test_pwcet_etp
}

pwcetIndepCase2 <- function(result, etpDownSampling)
{
  f1 = result$f1_ee
  f1Ret = result$f1_ret
  f2 = result$f2_ee
  f2Ret = result$f2_ret
  f3 = result$f3_ee
  f3Ret = result$f3_ret
  f4 = result$f4_ee
  f4Ret = result$f4_ret
  f5 = result$f5_ee
  f5Ret = result$f5_ret
  f6 = result$f6_ee
  f6Ret = result$f6_ret
  f7 = result$f7_ee
  f7Ret = result$f7_ret

  f1 = f1[ which(!f1 == 0)]
  f1Ret = f1Ret[ which(!f1Ret == 0)]
  f2 = f2[ which(!f2 == 0)]
  f2Ret = f2Ret[ which(!f2Ret == 0)]
  f3 = f3[ which(!f3 == 0)]
  f3Ret = f3Ret[ which(!f3Ret == 0)]
  f4 = f4[ which(!f4 == 0)]
  f4Ret = f4Ret[ which(!f4Ret == 0)]
  f5 = f5[ which(!f5 == 0)]
  f5Ret = f5Ret[ which(!f5Ret == 0)]
  f6 = f6[ which(!f6 == 0)]
  f6Ret = f6Ret[ which(!f6Ret == 0)]
  f7 = f7[ which(!f7 == 0)]
  f7Ret = f7Ret[ which(!f7Ret == 0)]

  test_entry = etp(as.vector(result$test_entry))
  f1_endtoend = etp(as.vector(f1))
  f1_ret = etp(as.vector(f1Ret))
  f2_endtoend = etp(as.vector(f2))
  f2_ret = etp(as.vector(f2Ret))
  f3_endtoend = etp(as.vector(f3))
  f3_ret = etp(as.vector(f3Ret))
```

```
    f4_endtoend = etp(as.vector(f4))
    f4_ret = etp(as.vector(f4Ret))
    f6_endtoend = etp(as.vector(f6))
    f6_ret = etp(as.vector(f6Ret))
    f7_endtoend = etp(as.vector(f7))
    f7_ret = etp(as.vector(f7Ret))

    test_self_pwcet_etp = indepConv(reduce_etp(test_entry, etpDownSampling),
        reduce_etp(f1_ret, etpDownSampling))
    test_self_pwcet_etp = indepConv(reduce_etp(test_self_pwcet_etp,
        etpDownSampling),reduce_etp(f2_ret, etpDownSampling))
    f3_ret_pwcet_etp_in_testProgram = f3_ret
    f4_ret_pwcet_etp_in_testProgram = f4_ret
    for (i in 2:5)
    {
      f3_ret_pwcet_etp_in_testProgram = indepConv(reduce_etp(f3_ret_pwcet_etp
          _in_testProgram, etpDownSampling),
                                                reduce_etp(f3_ret,
                                                    etpDownSampling))
      f4_ret_pwcet_etp_in_testProgram = indepConv(reduce_etp(f4_ret_pwcet_etp
          _in_testProgram, etpDownSampling),
                                                reduce_etp(f4_ret,
                                                    etpDownSampling))
    }
    test_self_pwcet_etp = indepConv(reduce_etp(test_self_pwcet_etp,
        etpDownSampling),
                                reduce_etp(f3_ret_pwcet_etp_in_
                                    testProgram, etpDownSampling))
    test_self_pwcet_etp = indepConv(reduce_etp(test_self_pwcet_etp,
        etpDownSampling),
                                reduce_etp(f4_ret_pwcet_etp_in_
                                    testProgram, etpDownSampling))
    test_self_pwcet_etp = indepConv(reduce_etp(test_self_pwcet_etp,
        etpDownSampling),reduce_etp(f6_ret, etpDownSampling))

    f1_pwcet_etp = f1_endtoend
    f2_pwcet_etp = f2_endtoend
    f3_pwcet_etp = f3_endtoend
    f4_pwcet_etp = f4_endtoend
    f6_pwcet_etp = f6_endtoend

    f3_pwcet_etp_in_testProgram = f3_pwcet_etp
    f4_pwcet_etp_in_testProgram = f4_pwcet_etp
    for (i in 2:5)
    {
      f3_pwcet_etp_in_testProgram = indepConv(reduce_etp(f3_pwcet_etp_in_
          testProgram, etpDownSampling),
                                            reduce_etp(f3_pwcet_etp,
                                                etpDownSampling))
      f4_pwcet_etp_in_testProgram = indepConv(reduce_etp(f4_pwcet_etp_in_
          testProgram, etpDownSampling),
                                            reduce_etp(f4_pwcet_etp,
                                                etpDownSampling))
    }

    test_pwcet_etp = indepConv(reduce_etp(test_self_pwcet_etp,
        etpDownSampling),reduce_etp(f1_pwcet_etp, etpDownSampling))
    test_pwcet_etp = indepConv(reduce_etp(test_pwcet_etp, etpDownSampling),
        reduce_etp(f2_pwcet_etp, etpDownSampling))
    test_pwcet_etp = indepConv(reduce_etp(test_pwcet_etp, etpDownSampling),
        reduce_etp(f3_pwcet_etp_in_testProgram, etpDownSampling))
    test_pwcet_etp = indepConv(reduce_etp(test_pwcet_etp, etpDownSampling),
        reduce_etp(f4_pwcet_etp_in_testProgram, etpDownSampling))
    test_pwcet_etp = indepConv(reduce_etp(test_pwcet_etp, etpDownSampling),
        reduce_etp(f6_pwcet_etp, etpDownSampling))
    test_pwcet_etp = reduce_etp(test_pwcet_etp, etpDownSampling)

    test_pwcet_etp
  }

  data <- readMat("/Users/lvnt/Desktop/captures/program_memman_loadraw.mat")

  result <- convertDataCase2(data)

  testEE = result$test_ee
```

```
testProgramEntry = result$test_entry
f1 = result$f1_ee
f1Ret = result$f1_ret
f2 = result$f2_ee
f2Ret = result$f2_ret
f3 = result$f3_ee
f3Ret = result$f3_ret
f4 = result$f4_ee
f4Ret = result$f4_ret
f5 = result$f5_ee
f5Ret = result$f5_ret
f6 = result$f6_ee
f6Ret = result$f6_ret
f7 = result$f7_ee
f7Ret = result$f7_ret

loop_cnt = result$loop_cnt
loop_sub = result$loop_ee_cum
loop_self = result$loop_ret_cum

cond1_sub = rbind(f2, loop_sub, f6, f7)
cond1_sub = colSums(cond1_sub)

cond1_self = rbind(f2Ret, loop_self, f6Ret, f7Ret)
cond1_self = colSums(cond1_self)

testProgramSelf = testProgramEntry + f1Ret + cond1_self
testProgramSub = f1 + cond1_sub

f2 = f2[ which(!f2 == 0)]
f2Ret = f2Ret[ which(!f2Ret == 0)]
f3 = f3[ which(!f3 == 0)]
f3Ret = f3Ret[ which(!f3Ret == 0)]
f4 = f4[ which(!f4 == 0)]
f4Ret = f4Ret[ which(!f4Ret == 0)]
f6 = f6[ which(!f6 == 0)]
f6Ret = f6Ret[ which(!f6Ret == 0)]
loop_sub = loop_sub[ which(!loop_sub == 0)]
loop_self = loop_self[ which(!loop_self == 0)]

df_testProgram <- data.frame(u = pobs(testProgramSelf), v = pobs(
    testProgramSub))
df_testProgramSub <- data.frame(u = pobs(f1), v = pobs(cond1_sub))
df_testProgramSelf <- data.frame(u = pobs(testProgramEntry), v = pobs(f1Ret
    ), k = pobs(cond1_self))
df_cond1if <- data.frame(u = pobs(f2), v = pobs(loop_sub), k = pobs(f6))
df_cond1ifret <- data.frame(u = pobs(f2Ret), v = pobs(loop_self), k = pobs(
    f6Ret))
df_loopBody <- data.frame(u = pobs(f3), v = pobs(f4))
df_loopBodyRet <- data.frame(u = pobs(f3Ret), v = pobs(f4Ret))

#chart.Correlation(df_testProgram, histogram=TRUE, pch=19)
#chart.Correlation(df_testProgramSub, histogram=TRUE, pch=19)
#chart.Correlation(df_testProgramSelf, histogram=TRUE, pch=19)
#chart.Correlation(df_cond1if, histogram=TRUE, pch=19)
#chart.Correlation(df_cond1ifret, histogram=TRUE, pch=19)
#chart.Correlation(df_loopBody, histogram=TRUE, pch=19)
#chart.Correlation(df_loopBodyRet, histogram=TRUE, pch=19)

RVM_testProgram <- RVineStructureSelect(df_testProgram, c(1:6), indeptest =
     TRUE)
RVM_testProgramSub <- RVineStructureSelect(df_testProgramSub, c(1:6),
    indeptest = TRUE)
RVM_testProgramSelf <- RVineStructureSelect(df_testProgramSelf, c(1:6),
    indeptest = TRUE)
RVM_cond1if <- RVineStructureSelect(df_cond1if, c(1:6), indeptest = TRUE)
RVM_cond1ifret <- RVineStructureSelect(df_cond1ifret, c(1:6), indeptest =
    TRUE)
RVM_loopBody <- RVineStructureSelect(df_loopBody, c(1:6), indeptest = TRUE)
RVM_loopBodyRet <- RVineStructureSelect(df_loopBodyRet, c(1:6), indeptest =
     TRUE)

#RVineGofTest(df_testProgram, RVM_testProgram, method="ECP2", statistic = "
    CvM")
```

```
#RVineGofTest(df_testProgramSub, RVM_testProgramSub, method="ECP2",
    statistic = "CvM")
#RVineGofTest(df_testProgramSelf, RVM_testProgramSelf, method="ECP2",
    statistic = "CvM")
#RVineGofTest(df_cond1if, RVM_cond1if, method="ECP2", statistic = "CvM")
#RVineGofTest(df_cond1ifret, RVM_cond1ifret, method="ECP2", statistic = "
    CvM")
#RVineGofTest(df_loopBody, RVM_loopBody, method="ECP2", statistic = "CvM")
#RVineGofTest(df_loopBodyRet, RVM_loopBodyRet, method="ECP2", statistic = "
    CvM")

simdata_testProgram <- RVineSim(1000000, RVM_testProgram)
simdata_testProgramSub <- RVineSim(1000000, RVM_testProgramSub)
simdata_testProgramSelf <- RVineSim(1000000, RVM_testProgramSelf)
simdata_cond1if <- RVineSim(1000000, RVM_cond1if)
simdata_cond1ifret <- RVineSim(1000000, RVM_cond1ifret)
simdata_loopBody <- RVineSim(1000000, RVM_loopBody)
simdata_loopBodyRet <- RVineSim(1000000, RVM_loopBodyRet)

f1_cdf = etp(as.vector(f1))
f1_cdf[,2] = cumsum(f1_cdf[,2])

f1Ret_cdf = etp(as.vector(f1Ret))
f1Ret_cdf[,2] = cumsum(f1Ret_cdf[,2])

f2_cdf = etp(as.vector(f2))
f2_cdf[,2] = cumsum(f2_cdf[,2])

f2Ret_cdf = etp(as.vector(f2Ret))
f2Ret_cdf[,2] = cumsum(f2Ret_cdf[,2])

f3_cdf = etp(as.vector(f3))
f3_cdf[,2] = cumsum(f3_cdf[,2])

f3Ret_cdf = etp(as.vector(f3Ret))
f3Ret_cdf[,2] = cumsum(f3Ret_cdf[,2])

f4_cdf = etp(as.vector(f4))
f4_cdf[,2] = cumsum(f4_cdf[,2])

f4Ret_cdf = etp(as.vector(f4Ret))
f4Ret_cdf[,2] = cumsum(f4Ret_cdf[,2])

f6_cdf = etp(as.vector(f6))
f6_cdf[,2] = cumsum(f6_cdf[,2])

f6Ret_cdf = etp(as.vector(f6Ret))
f6Ret_cdf[,2] = cumsum(f6Ret_cdf[,2])

f7_cdf = etp(as.vector(f7))
f7_cdf[,2] = cumsum(f7_cdf[,2])

f7Ret_cdf = etp(as.vector(f7Ret))
f7Ret_cdf[,2] = cumsum(f7Ret_cdf[,2])

-----------------------------------loopBody
    -----------------------------------
f3_real = findInterval(simdata_loopBody[,1], f3_cdf[,2])+1
f3_real = f3_cdf[,1][f3_real]

f3Ret_real = findInterval(simdata_loopBodyRet[,1], f3Ret_cdf[,2])+1
f3Ret_real = f3Ret_cdf[,1][f3Ret_real]

f4_real = findInterval(simdata_loopBody[,2], f4_cdf[,2])+1
f4_real = f4_cdf[,1][f4_real]

f4Ret_real = findInterval(simdata_loopBodyRet[,2], f4Ret_cdf[,2])+1
f4Ret_real = f4Ret_cdf[,1][f4Ret_real]

loopBody_W = f3_real + f4_real
loopBody_etp = etp(as.vector(loopBody_W))

loopBodyRet_W = f3Ret_real + f4Ret_real
loopBodyRet_etp = etp(as.vector(loopBodyRet_W))
```

```
-----------------------------------loopBody
    -----------------------------------

-----------------------------------loopSub-----------------------------------
loopBody_etp_tmp = reduce_etp(loopBody_etp, 1000) #do not call this when
    biased conv. is applied
loop_sub_cdf = loopBody_etp_tmp

for (i in 2:max(loop_cnt))
{
  loop_sub_cdf = reduce_etp(loop_sub_cdf, 1000)
  loop_sub_cdf = indepConv(loop_sub_cdf,loopBody_etp_tmp)
}
loop_sub_cdf[,2] = cumsum(loop_sub_cdf[,2])

loopBodyRet_etp_tmp = reduce_etp(loopBodyRet_etp, 1000) #do not call this
    when biased conv. is applied
loop_self_cdf = loopBodyRet_etp_tmp

for (i in 2:max(loop_cnt))
{
  loop_self_cdf = reduce_etp(loop_self_cdf, 1000)
  loop_self_cdf = indepConv(loop_self_cdf,loopBodyRet_etp_tmp)
}
loop_self_cdf[,2] = cumsum(loop_self_cdf[,2])
-----------------------------------loopSub-----------------------------------

-----------------------------------cond1if-----------------------------------
f2_real = findInterval(simdata_cond1if[,1], f2_cdf[,2])+1
f2_real = f2_cdf[,1][f2_real]

f2Ret_real = findInterval(simdata_cond1ifret[,1], f2Ret_cdf[,2])+1
f2Ret_real = f2Ret_cdf[,1][f2Ret_real]

loop_sub_real = findInterval(simdata_cond1if[,2], loop_sub_cdf[,2])+1
loop_sub_real = loop_sub_cdf[,1][loop_sub_real]

loop_self_real = findInterval(simdata_cond1ifret[,2], loop_self_cdf[,2])+1
loop_self_real = loop_self_cdf[,1][loop_self_real]

f6_real = findInterval(simdata_cond1if[,3], f6_cdf[,2])+1
f6_real = f6_cdf[,1][f6_real]

f6Ret_real = findInterval(simdata_cond1ifret[,3], f6Ret_cdf[,2])+1
f6Ret_real = f6Ret_cdf[,1][f6Ret_real]

cond1if_W = f2_real + loop_sub_real + f6_real
cond1if_cdf = etp(as.vector(cond1if_W))
cond1if_cdf[,2] = cumsum(cond1if_cdf[,2])

cond1ifret_W = f2Ret_real + loop_self_real + f6Ret_real
cond1ifret_cdf = etp(as.vector(cond1ifret_W))
cond1ifret_cdf[,2] = cumsum(cond1ifret_cdf[,2])
-----------------------------------cond1if-----------------------------------

-----------------------------------testProgramSub
    -----------------------------------
f1_real = findInterval(simdata_testProgramSub[,1], f1_cdf[,2])+1
f1_real = f1_cdf[,1][f1_real]

f1Ret_real = findInterval(simdata_testProgramSelf[,1], f1Ret_cdf[,2])+1
f1Ret_real = f1Ret_cdf[,1][f1Ret_real]

condif_real = findInterval(simdata_testProgramSub[,2], cond1if_cdf[,2])+1
condif_real = cond1if_cdf[,1][condif_real]

condifRet_real = findInterval(simdata_testProgramSelf[,2], cond1ifret_cdf
    [,2])+1
condifRet_real = cond1ifret_cdf[,1][condifRet_real]

f7_real = findInterval(simdata_testProgramSub[,2], f7_cdf[,2])+1
f7_real = f7_cdf[,1][f7_real]

f7Ret_real = findInterval(simdata_testProgramSelf[,2], f7Ret_cdf[,2])+1
f7Ret_real = f7Ret_cdf[,1][f7Ret_real]
```

```
cond1_sub_W = apply(cbind(condif_real, f7_real), 1, max)

cond1_self_W = apply(cbind(condifRet_real, f7Ret_real), 1, max)

testProgramSub_W = f1_real + cond1_sub_W
testProgramSub_cdf = etp(as.vector(testProgramSub_W))
testProgramSub_cdf[,2] = cumsum(testProgramSub_cdf[,2])

testProgramSelf_W = f1Ret_real + cond1_self_W
testProgramSelf_cdf = etp(as.vector(testProgramSelf_W))
testProgramSelf_cdf[,2] = cumsum(testProgramSelf_cdf[,2])
----------------------------------testProgramSub
    ----------------------------------

testProgramSelf_real = findInterval(simdata_testProgram[,1],
    testProgramSelf_cdf[,2])+1
testProgramSelf_real = testProgramSelf_cdf[,1][testProgramSelf_real]

testProgramSub_real = findInterval(simdata_testProgram[,2], testProgramSub_
    cdf[,2])+1
testProgramSub_real = testProgramSub_cdf[,1][testProgramSub_real]

testProgram_W = testProgramSelf_real + testProgramSub_real




testProgram_ee_etp = etp(as.vector(testEE))
testProgram_spdCop_etp = etp(as.vector(testProgram_W))
testProgram_biasedConv_etp = pwcetComonCase2(result, 0)
testProgram_independent_etp = pwcetIndepCase2(result, 10000)

ee_x = testProgram_ee_etp[,1]
ee_y = eecdf(testProgram_ee_etp)[,2]

spdcop_x = testProgram_spdCop_etp[,1]
spdcop_y = eecdf(testProgram_spdCop_etp)[,2]

comon_x = testProgram_biasedConv_etp[,1]
comon_y = eecdf(testProgram_biasedConv_etp)[,2]

indep_x = testProgram_independent_etp[,1]
indep_y = eecdf(testProgram_independent_etp)[,2]

p <- plot_ly(x = ee_x, y = ee_y, type = 'scatter', mode = 'lines', name = '
    EE') %>%
      add_trace(x = spdcop_x, y = spdcop_y, type = 'scatter', mode = 'lines
          ', name = 'EVT-COP') %>%
      add_trace(x = comon_x, y = comon_y, type = 'scatter', mode = 'lines',
           name='RapiTime') %>%
      add_trace(x = indep_x, y = indep_y, type = 'scatter', mode = 'lines',
           name='Independent')
p
layout(p, yaxis = list(type = "log", exponentformat="power", showexponent="
    all"))

testProgram_ee_etp[,2] = cumsum(testProgram_ee_etp[,2])
testProgram_spdCop_etp[,2] = cumsum(testProgram_spdCop_etp[,2])
testProgram_biasedConv_etp[,2] = cumsum(testProgram_biasedConv_etp[,2])
testProgram_independent_etp[,2] = cumsum(testProgram_independent_etp[,2])

testProgram_spdCop_etp[,1][findInterval(0.99, testProgram_spdCop_etp[,2])
    +1] / 200000
testProgram_independent_etp[,1][findInterval(0.99, testProgram_independent_
    etp[,2])+1] / 200000
testProgram_biasedConv_etp[,1][findInterval(0.99, testProgram_biasedConv_
    etp[,2])+1] / 200000
testProgram_ee_etp[,1][findInterval(0.99, testProgram_ee_etp[,2])+1] /
    200000

testProgram_spdCop_etp[,1][findInterval(0.9999, testProgram_spdCop_etp[,2])
    +1] / 200000
```

```
testProgram_independent_etp[,1][findInterval(0.9999, testProgram_
    independent_etp[,2])+1] / 200000
testProgram_biasedConv_etp[,1][findInterval(0.9999, testProgram_biasedConv_
    etp[,2])+1] / 200000
testProgram_ee_etp[,1][findInterval(0.9999, testProgram_ee_etp[,2])+1] /
    200000

testProgram_spdCop_etp[,1][findInterval(0.999999, testProgram_spdCop_etp
    [,2])+1] / 200000
testProgram_independent_etp[,1][findInterval(0.999999, testProgram_
    independent_etp[,2])+1] / 200000
testProgram_biasedConv_etp[,1][findInterval(0.999999, testProgram_
    biasedConv_etp[,2])+1] / 200000
testProgram_ee_etp[,1][findInterval(0.999999, testProgram_ee_etp[,2])+1] /
    200000

testProgram_spdCop_etp[,1][findInterval(0.999999999, testProgram_spdCop_etp
    [,2])+1] / 200000
testProgram_independent_etp[,1][findInterval(0.999999999, testProgram_
    independent_etp[,2])+1] / 200000
testProgram_biasedConv_etp[,1][findInterval(0.999999999, testProgram_
    biasedConv_etp[,2])+1] / 200000
testProgram_ee_etp[,1][findInterval(0.999999999, testProgram_ee_etp[,2])+1]
     / 200000
```