POLITECNICO DI MILANO
SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

TESI DI LAUREA MAGISTRALE IN COMPUTER SCIENCE AND ENGINEERING

DEVELOPMENT AND EVALUATION OF WCET BENCHMARKS FOR PROBABILISTIC REAL-TIME APPROACHES

Author:
Dott. Riccardo Confalonieri
Student ID:
920410

Supervisor:
Prof. William Fornaciari

Co-Supervisor (Correlatore):
Dott. Federico Reghenzani

Academic Year – 2019/20

Contents

List of Figures

List of Tables

Acronyms

BM Block Maxima.

CDF Cumulative Distribution Function.

COTS Commercial Off-The-Shelf.

CV Coefficient of Variation.

EVT Extreme Value Theory.

GEVD Generalized Extreme Value Distribution.

GoF Goodness-of-Fit.

GPD Generalized Pareto Distribution.

GPIO General Purpose Input/Output.

HDTA Hybrid Deterministic Timing Analysis.

HPC High-Performance Computing.

HRT High-Resolution Timer.

i.i.d. Independent and Identically Distributed.

IRQ Interrupt Request.

Acronyms

MBDTA Measurement-Based Deterministic Timing Analysis.

MDA Maximum Domain of Attraction.

MLE Maximum Likelihood Estimator.

OS Operating System.

PNRG Pseudo-Random Number Generator.

PoT Peak Over Threshold.

PPI Probabilistic Predictability Index.

pWCET Probabilistic Worst-Case Execution Time.

PWM Probability Weighted Moments.

RCU Read-Copy-Update.

RT Real-Time.

SDTA Static Deterministic Timing Analysis.

SPTA Static Probabilistic Timing Analysis.

TCAS Traffic Alert and Collision Avoidance System.

WCET Worst-Case Execution Time.

Acknowledgments

Il conseguimento di questo importante risultato ha richiesto impegno, dedizione e sacrifici. Non sarei però mai riuscito a raggiungere questo traguardo senza l'aiuto di chi, durante il percorso, mi è stato vicino e mi ha aiutato.

Ringrazio innanzitutto la mia famiglia. A Lara, per aver condiviso con me le gioie e le difficoltà della vita da universitari, per la capacità di distrarmi dallo studio tutte le volte che ne avevo bisogno e per avermi spinto a credere di più in me stesso. Ai miei genitori e a mia sorella, per avermi supportato durante tutti gli studi, non solo economicamente, ma soprattutto moralmente e per avermi sempre assecondato in ogni mia scelta. Ai miei nonni, per l'orgoglio dimostratomi ad ogni esame superato.

Grazie poi ai miei compagni di corso, da quelli rimasti solo per poco tempo, a chi ha condiviso con me l'intero percorso: i lavori di gruppo e i progetti svolti con voi sono stati di grande aiuto per vincere la mia timidezza. Ringrazio in particolare Valerio, Samuele e Simone, che da compagni sono diventati amici.

Grazie infine al professor William Fornaciari e al correlatore Federico Reghenzani, per non avermi mai fatto mancare il loro appoggio nonostante la situazione anomala che abbiamo dovuto affrontare durante la stesura di questa tesi.

Abstract

The accurate estimation of the Worst-Case Execution Time (WCET) is essential when dealing with hard Real-Time systems, in particular when they execute mission-critical or safety-critical applications. Traditional static methods for its computation are, however, not efficient when used on modern hardware architectures, especially in multi-core and many-core CPUs. In this scenario, a promising solution is represented by Probabilistic Timing Analysis, aiming at determining the probabilistic-WCET. Current WCET benchmark suites have been developed mainly for static analyses, lacking information needed to test probabilistic techniques. This thesis recaps the currently available WCET benchmarks, with an in-depth analysis of their statistical characteristics, focusing, in particular, on probabilistic-WCET properties. Then, we propose possible modifications to these benchmarks to improve the applicability with probabilistic real-time, as well as novel benchmarks created from scratch. Several experiments, executed on a STM32L010RB microcontroller and a Raspberry PI 4, assess the predictability of the benchmarks in different configurations, allowing us to draw advice to be used in future academic works and industrial applications.

Abstract (in Italiano)

Un'accurata stima del Worst-Case Execution Time (WCET) è essenziale quando si ha a che fare con sistemi hard Real-Time, in particolar modo quando questi eseguono applicazioni di tipo mission-critical o safety-critical. Tuttavia, i tradizionali metodi di calcolo non sono efficienti se utilizzati con le più moderne architetture hardware, specialmente su CPU multi-core e many-core. In questi casi, una soluzione promettente è rappresentata dalla Probabilistic Timing Analysis, il cui obiettivo è la determinazione del probabilistic-WCET. Le attuali suites di benchmarks per il WCET sono state sviluppate principalmente per l'analisi statica e mancano quindi di informazioni necessarie a testare le tecniche probabilistiche. Questa tesi riepiloga i benchmarks per il WCET attualmente esistenti, con un'analisi specifica delle loro caratteristiche statistiche, concentrandosi, in particolare, sulle proprietà relative al probabilistic-WCET. In seguito, proponiamo possibili modifiche ai benchmarks, per migliorare la loro applicabilità alle tecniche di probabilistic real-time, insieme a benchmarks creati interamente da zero. Diversi esperimenti, condotti su un microcontrollore STM32L010RB e su una Raspberry PI 4, valutano la predicibilità dei benchmarks con diverse configurazioni, permettendoci di definire alcune linee guida da usare per la futura ricerca accademica e per l'applicazione industriale.

CHAPTER 1

Introduction

1.1 Embedded systems and Real-Time systems

Embedded systems are computing systems where hardware and software integration are tightly coupled. They are usually part of a bigger system (the embedding one), where they may cooperate with other embedded systems in a distributed environment, but some of them can also work on their own. They are designed to perform specific tasks and to optimize specific parameters such as size, cost of production, and power consumption, while guaranteeing very high levels in terms of performance and reliability. Nowadays, smart and interconnected embedded systems are getting more and more present not only in electronic devices, such as smartwatches and cameras, but also on household appliances used in everyday life, like fridges, washing machines, and televisions. Another sector in which this kind of devices' employment is rapidly increasing is automotive: statisticians forecast about 50% of the cost of cars will be due to the electronic devices in 2030, as shown in Figure 1.1.

A subset of embedded systems is different from the rest due to the

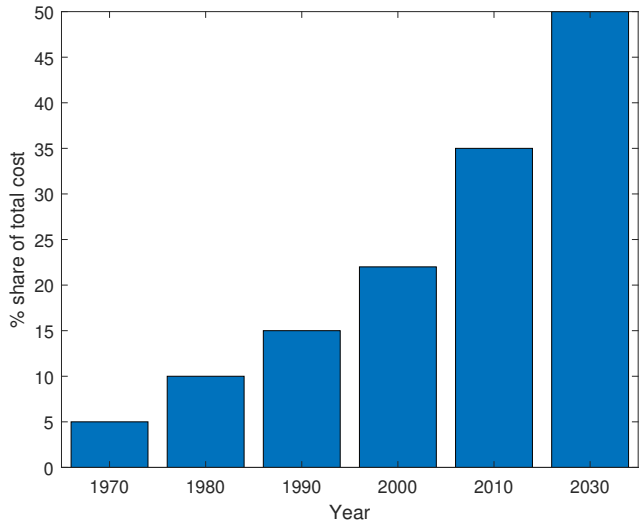


Figure 1.1: *The evolution of the impact of embedded systems' cost on the total cost of cars.*

requirement of reacting to external events within a specified amount of time: *Real-Time systems*. This kind of devices can be further split into two categories: in *hard* Real-Time systems, missing a timing constraint determines the failure of the whole system, while in *soft* Real-Time systems, delays cause degradation in the quality of service, but are still acceptable, provided that their frequency is not too high. Examples of soft Real-Time systems are videogames, for which a brief reduction of the video frame rate makes the experience of the player worse, but does not prevent him from keeping on playing, or software for live audio and/or video conferences, in which missing some brief part of the communication causes the quality of the call to be lower, but still allows users to continue their meeting after that. In both cases, if the time interval in which the reduction of the Quality of Service occurs is short enough, the users may even not notice the degradation. Instead, examples of hard Real-Time systems are the navigation systems of aircraft, fire protection and alarm systems, online trading systems of banks, control systems of nuclear plants, or the software used to switch on the engine of a car. In these cases, instead, even a small violation of the deadline may cause the system to misbehave. Usually, hard real-time systems are also critical systems, which can be further categorized in *mission-critical* and *safety-critical*. Among the previously

shown examples, only trading systems and cars' engine software belong to the first category. In fact, non-operating online systems make it impossible for the customers of the bank to rapidly buy or sell stocks on the market (with possible loss of money for them, or for the bank itself), and the failure of the engine software may prevent the car to switch on, or even damage the engine. In any case, apart from the potential economic loss, none of the two malfunctions of such applications constitutes a danger for anyone. All the other systems are instead examples of safety-critical applications, since having them not respecting their deadlines may cause people to get injured, or even worse to lose their lives, as in the case of a lately reported fire, the control system of a plane failing to react in time, or the meltdown of a nuclear power plant. In these cases, it is essential to guarantee not only the functional correctness of the software, but also the timing correctness, i.e. the ability for the tasks to meet all the timing deadlines.

1.2 The WCET problem

The estimation of the *Worst-Case Execution Time* (WCET) is essential for hard real-time systems, in which the timing constraints of the tasks must be always satisfied. Failing to meet these constraints leads the system to behave improperly, with possible unacceptable consequences, especially in the case of mission-critical or safety-critical applications, as previously described. Furthermore, accurate WCET estimations can introduce improvements also at the level of run-time resource management. Some policies of resource allocation, in fact, may gain advantage from the possibility of knowing in advance the amount of computing resources needed to be reserved to a task which is ready to be executed, in order to guarantee its completion within a given time [26]. This would implicitly minimize the resource over-provisioning, and thus costs, maximizing at the same time the executable workload. For these reasons, a timing analysis requires the estimated WCET value to be greater or equal than the real WCET. Also, this estimation must be as tight as possible to the real WCET, in order to minimize the amount of resources assigned to a specific task without necessity.

1.2.1 The hardware complexity

During the past two decades, the hardware platforms used in real-time systems have become increasingly more complex due to both the increasing computational power demand and, at the same time, the limitations in single-core power and performance. In fact, the performance limit in the development of single-core CPUs has been reached, making it pointless to focus on further development of the single-core computational power, and thus forcing the employment of multi-core solutions in the vast majority of the applications. One of the main reasons why this happened was the end of *Moore's law* validity. Unlike the name would suggest, it is not a law of physics, but an empirical observation based on the experience of Gordon Moore, the co-founder of *Fairchild Semiconductor* and *Intel*, who in 1965 forecasted the number of transistors in a dense integrated circuit to double every year. His prediction was valid for 10 years, and in 1975 it was revised by Moore himself, who expected the number to double every two years from that point on. The new law has applied until 2010, when an industry-wide slow down below the predicted rate was noticed.

Another cause of the reaching of the limit in the single-core performance was the breakdown of *Dennard scaling*, a scaling law formulated in 1974 in [6], a paper whose Robert H. Dennard was one of the authors. It stated that, in an electric circuit, power density is constant, thus energy consumption grows proportionally with the area. According to his calculations, dimensions of the transistors could have been decreased by 30% every technology generation (thus reducing also delays and voltage by the same percentage, allowing a 40% increase in operating frequency, while reducing power by 50%), resulting in a 50% reduction in the total area. Summing up, an evolution doubling the density of the transistors brings to a speedup of 40%, while keeping power consumption unchanged. This law was valid until about 2006, when improvements in the technology began to result in lower frequency increase than expected and, thus, an increasing power density.

Nowadays, architectures also include advanced optimization techniques such as pipelines, branch prediction, out-of-order execution, caches, writebuffers, scratchpads, and multiple levels of memory hierarchy. Adding complexity makes it harder to create reliable models of the processors. This problem is even amplified when considering previously mentioned

multi-core and many-core platforms, with different tasks running at the same time and producing interferences which are very hard to predict and analyse. Furthermore, most of the acceleration features are *Commercial Off The Shelf* (COTS), designed to optimize the average-case rather than the worst-case behaviour, and can result in significant variability in execution times. They are very common in non-critical industrial world (unless high customisation level, requiring ad-hoc components, is needed), since they allow to reduce development costs and require very low maintenance effort, usually being already very well tested by the producer, who can base on feedbacks and issues reported by all the customers.

The increasing software and hardware complexity makes it difficult to define measurement protocols able to ensure that the worst-case path(s) through the code are visited, and that the worst-case hardware states are taken into account during the analysis. For all these reasons, traditional methodologies of static WCET analyses fail to obtain safe and tight WCET estimations when dealing with modern platforms, in a sustainable amount of time, since the required computational complexity would be unfeasible, or many approximations should be introduced, that generates a very pessimistic WCET. Consequently, researchers are searching for new techniques to overcome these issues.

1.3 Timing analyses techniques

1.3.1 Traditional techniques

Traditional analyses techniques aim at finding a single value of the WCET, i.e. a value upper-bounding all execution times obtainable with every possible input and hardware state. As shown by Abella et al. [1], these methods can be divided into three categories:

- **Static Deterministic Timing Analyses** (SDTA): WCET estimation is obtained by just analysing the code, without actually executing it. First, a control flow analysis is performed to find out which are the feasible paths and the bounds of the loops in the program. Then, the focus moves to the hardware architecture, in order to calculate an over-approximation of the previously discovered paths by taking into account factors such as caches, pipelines, and memory structure. Finally, integer linear programming is employed to combine the

results of the first two steps. Of course, the main challenge for this kind of analysis to provide sound outcome is to have very accurate estimations both on hardware and software sides, which becomes more and more tricky with the growth of the complexity of the code and architecture. For instance, it may not be easy to determine the content of a cache at a given point (and thus the rate of hits and misses) when input values show some kind of correlation;

- **Measurement-Based Deterministic Timing Analyses** (MBDTA): the execution time of the task is measured by testing different configurations, both in terms of input and initial hardware state. The highest obtained time can then be taken as lower bound for WCET computation, or directly used for the calculation of the WCET itself by adding some overhead, whose value can be fixed, or better a fraction of the actual measured time. Of course, for the estimation to be more accurate, inputs leading to the highest execution times should be included in the test vector, as well as the worst hardware states. Anyway, finding out the configurations making the highest times occur becomes more and more difficult with the increasing of the complexity of both the programs and the hardware they are tested on;

- **Hybrid Deterministic Timing Analyses** (HDTA): it is the technique in which the previous two are combined, trying to solve the weaknesses of both of them. In this case, time is measured only on sub-paths of the program, that makes finding the input leading to the worst-case path not necessary anymore. Then results are combined through static analysis techniques, that can now be considered only for what concerns control flow analysis, while studies about hardware structure can be avoided (since execution times of the different parts of the program have already been empirically measured), which is the biggest advantage of this method. The issue, instead, is that the execution time of each sub-task is very likely to be influenced by the execution history, and thus by previously executed sub-tasks, which are not considered in this method.

1.3.2 Probabilistic timing analysis

Probabilistic timing analyses differ from traditional approaches in the fact that they move the characterisation of the timing behaviour of a program from taking into account just a single run of the program itself, to the consideration of a repeating sequence of many runs, referred to as a *scenario*, thus changing the results from a scalar value –the WCET– to a probability distribution –the *probabilistic-WCET distribution* (pWCET). While traditional timing analysis methods aim to tightly upper bound the execution time that could occur for a single run of a program out of all possible runs, probabilistic timing analysis methods aim to tightly upper bound the distribution of execution times that could potentially occur for some scenario of operation, out of all possible scenarios of operation. The techniques of analysis belonging to this category are:

- **Static Probabilistic Timing Analysis** (SPTA): as it happened in SDTA, the execution of the program on the actual hardware or a simulator is not directly measured. The difference is that in SPTA random phenomena in the environment (such as cache state, rate of correctly predicted branches, or input correlation) are modelled through probability distributions. Thus, the result of the analysis is an upper bound of the pWCET, and not just a single value upper-bounding the WCET. Anyway, the problems with this method are similar to the corresponding deterministic one, since a poor approximations of very complex dynamics can lead to a large over-estimation of the actual pWCET. This is the reason why this method is not very appealing for current research;

- **Measurement-Based Probabilistic Timing Analysis** (MBPTA): as in the corresponding deterministic technique, programs are actually run to measure their execution times. This can be done through two different approaches: *per-path*, in which the observations are separated according to the path traversed in the single execution, and *per-program*, where instead all the observations are grouped together to have a direct estimation of the pWCET. In both cases, for the estimation to be possible, collected results must pass proper statistical tests. From this perspective, the first method can give some advantage, since measurements made with the per-program

approach may fail the tests in case different paths lead to very different distributions, while in the per-path approach it happens just in case different inputs determine much different behaviour through the same path, which is less likely. Furthermore, the issue of representativity (i.e. test input must reflect the one that will actually occur during operation) is simpler when employing the per-path approach rather than per-program. Anyway, in the first case, the frequency at which the different paths are exercised has a lot of impact on the final pWCET estimate, and therefore it must be precisely estimated: for instance, visiting a path with high execution times more often during analysis than what will happen during the actual usage of the application will result in a pWCET over-estimate. Another disadvantage of the per-path approach is that by splitting measurements information about ordering and dependencies between consecutive executions, which may be very important when computing pWCET, is lost. In any case, the per-program approach is the one that is usually chosen, since we do not have to divide outcomes according to the path they derive from, making in this way the analysis more practical;

- **Hybrid Probabilistic Timing Analysis (HPTA)**: it is a mixed approach, combining the techniques belonging to the previous two.

1.4 Motivation and contribution of this thesis

1.4.1 Motivation

In the previous paragraphs, we showed how the problem of having a very accurate estimate of the WCET of a real-time application in mission-critical and safety-critical environments created the need for alternative WCET estimation techniques for modern systems. Probabilistic real-time, based on a so-called pWCET distribution, is a promising solution. The pWCET upper bounds all possible scenarios of operation, and not just a single value like the WCET. In order to obtain good confidence in pWCET estimations, probabilistic methods are used. Such tools require some hypotheses to be satisfied and verified via statistical procedures. Current state-of-the-art benchmarks have been used in previous works, but not appropriately analysed. For this reason, creating a suite of statistically

characterized benchmarks would be beneficial for the scientific community, in particular for pWCET analyses. A common and stable benchmark suite allows the comparisons between the different methods and architectures. The lack of such a collection, carefully studied for probabilistic purposes, is the motivation that inspired this thesis.

1.4.2 Contribution

The work of this thesis begins by changing the structure (but not the behaviour) of already existing programs, taken from some of the available benchmark suites for WCET, to be able to execute them under different execution conditions and inputs. We use the obtained results to perform statistical analyses, in particular related to the *probabilistic predictability index*, a value stating whether a certain program is suitable for probabilistic timing analysis. After the analysis of the current benchmarks, we modify them and create new benchmarks from scratch, in order to improve the satisfaction of the statistical hypotheses needed by pWCET approaches, and better represent real applications and situations. The experiments, executed on different architectures and by using different instruments for time measurements, allow us to make several considerations and comparisons about the different setups, which, in turn, lead to propose advice and best-practices for the use of probabilistic real-time in future works.

1.4.3 Structure

After this introduction, in which an overview of Embedded and Real-Time Systems is given, together with an explanation of the WCET problem and of the different timing analysis techniques, Chapter 2 presents the background knowledge which is needed to fully understand the content of this thesis, with a special focus on Probabilistic Timing Analysis. Chapter 3 is dedicated to the State of the Art: the first part is about scientifical papers on the development of MBPTA, while the second shows the characteristics of the available benchmark suites for WCET. Then, in Chapter 4, we move to the description of the experimental setup, including the different architectures and time-measuring instruments employed in the analysis. After that, we begin to show the experimental results we obtained with an *STM32 Nucleo Board*: Chapter 5 contains the results related to the original benchmarks, while Chapter 6 presents the results obtained

from the modified benchmarks and the brand new ones. In Chapter 7, code executed in the previous two chapters is re-run on a Raspberry PI, with the standard Linux version *Raspberry Pi OS 1.4* and the *PREEMPT_RT Patch* installed, and related results are shown. Finally, in Chapter 8, conclusions and possible future works related to this thesis are presented.

CHAPTER 2

Background knowledge

In this chapter, some background knowledge, needed to fully understand the content of this thesis, is provided. After introducing the concept of *pWCET*, we explain how this kind of distribution can be used, and why the application of the *Extreme Value Theory* is fundamental to reach high levels of accuracy. This last concept is deeply analysed, through a description of the two distributions that can be used to fit data, the procedure needed, and the hypotheses that must be checked in order to apply the algorithm. Then, we provide an overview of how statistical tests are performed, before introducing *PPI*, an index summarizing the outcomes of the different statistical tests.

2.1 Probabilistic Real-Time computing

The classification of timing analyses has been presented in Section 1.3. Among them, the probabilistic approaches are the Static Probabilistic Timing Analysis (SPTA) and the Measurement-Based Probabilistic Timing Analysis (MBPTA).

Probabilistic real-time, and in particular Measurement-Based Probabilistic Timing Analysis (MBPTA), has been introduced since the beginning of the 2000s. This technique can become very useful in the industrial world, because it enables the WCET computation with little effort and independently from the complexity of the hardware. This method estimates a statistical distribution of the WCET (pWCET) directly from the execution time observations. In this chapter, we will see how a very accurate estimate of the pWCET is needed, and how it can be used, together with an overview about *Extreme Value Theory* (EVT), the way it is applied to the pWCET problem, and software and hardware related issues that may cause its usage to not be possible.

2.1.1 The pWCET distribution

A pWCET distribution is defined as the tightest upper bound of the execution time distribution of a program for every feasible scenario of operation, which is an infinite sequence of input parameters and hardware internal states that may occur when repeatedly execute the program itself. It is convenient to write the pWCET distribution with its complementary cumulative distribution function:

$$p = 1 - F(\overline{WCET}) = P(X > \overline{WCET}),$$

where X is the random variable representing the program execution time, p the probability of observing execution times greater than $\overline{WCET}$, which is a constant, and $F(\cdot)$ the cumulative distribution function (cdf). In fact, having this cdf, it is possible to make two kinds of operations:

1. Fixing a value for $\overline{WCET}$ and compute the probability p of observing an execution time longer than this value;
2. Fixing a value for the probability p and compute the value $\overline{WCET}$ for which the probability p represents the probability of observing a longer execution time.

The goal of any timing analysis, in particular for safety-critical systems, is to obtain a safe and accurate estimation of the pWCET, in order to guarantee a sufficiently low probability of failure (e.g. 10^{-9}) able to meet the failure requirements. However, for such low probability values,

we need to collect a large number of timing measurements to achieve a sufficient level of confidence, which is clearly not practical. That is why we can exploit the **Extreme Value Theory (EVT)**, which has been developed to study the tails of the distributions and so it is positioned at the opposite to the famous *central limit theorem*, which instead focuses on the behaviour of the distribution around its mean value.

2.1.2 The Extreme Value Theory

Given a sequence of independent and identically distributed (i.i.d.) random variables $X_1, X_2, ..., X_n$, the EVT provides the limit distribution at the extremes, i.e. the $\max(X_1, X_2, ..., X_n)$ or $\min(X_1, X_2, ..., X_n)$. In the scenario we are considering in this thesis, $X_1, X_2, ..., X_n$ is a sequence of execution times of a given program. Consequently, since for the WCET estimation we are interested in the maximum value, we can formalize the probability of not observing an execution time longer than a certain value x as follows:

$$\begin{aligned} P(\max(X_1, X_2, ..., X_n) \leq x) &= P(X_1 \leq x, X_2 \leq x, ..., X_n \leq x) \\ &\overset{i.i.d.}{=} P(X_1 \leq x)P(X_2 \leq x) \cdots P(X_n \leq x) = F^n(x), \end{aligned}$$

where, as already mentioned above, $F^n(x)$ is the cumulative distribution function of $X_1, X_2, ..., X_n$. The most important result of the EVT is that there exists a sequence $a_n \in \mathbb{R}$ and a sequence b_n such that:

$$\lim_{n \to \infty} F^n(b_n + a_n x) = G(x)$$

where $G(x)$ is the cdf of a statistical distribution called extreme value distribution. The interesting result on this formula is the *Fisher-Tippett-Gnedenko theorem*, which says that if the previous limit is non-degenerating, $G(x)$ is a cdf of well-known distributions, independently on the form of $F(x)$. The original Fisher-Tippett-Gnedenko theorem provided three possible forms for $G(x)$: Weibull, Frechet, and Gumbell distributions. However, it has been later proved in [9] and [13] that they can be generalized in the

Generalized Extreme Value Distribution (GEVD) that has the following form:

$$G(x) = \begin{cases} e^{-e^{\frac{x-\mu}{\sigma}}} & \xi = 0 \\ e^{-[1+\xi(\frac{x-\mu}{\sigma})]^{-1/\xi}} & \xi \neq 0 \end{cases}$$

Another, but asymptotically equivalent, formulation for $G(x)$ is the **Generalized Pareto Distribution (GPD)** having the following form:

$$G(x) = \begin{cases} 1 - e^{\frac{x-\mu}{\sigma}} & \xi = 0 \\ 1 - [1+\xi(\frac{x-\mu}{\sigma})]^{-1/\xi} & \xi \neq 0 \end{cases}$$

Both distributions have three parameters: the location μ, the scale σ, and the shape ξ. While the first two parameters are a similar concept to the average and standard deviation, the third parameter has an important role when considering pWCET distributions. In fact, depending on the sign of this parameter, the shape of the tail of the distribution significantly changes. Talking about *Generalized Extreme Value Distribution*, if $\xi > 0$ the GEVD converges to the Frechet distribution, if $\xi < 0$ it converges to the Weibull distribution, otherwise if $\xi \to 0$ it converges to the Gumbel distribution. Instead, considering the *Generalized Pareto Distribution* if $\xi = 0$ the GPD converges to the exponential distribution, while if $\xi > 0$ and $\mu = \frac{\sigma}{\xi}$ it converges to the Pareto distribution. Two different methods exist to fit these distributions. The first is called **Block-Maxima (BM)** and can be used to fit GEVD: the set of observations is divided into blocks of size B (which is the parameter that must be correctly tuned here) and the maximum value in each block is taken. Formally, the following filter is applied:

$$Y_i = \max(X_{B\cdot(i-i)}, X_{B\cdot(i-i)+1}, ..., X_{B\cdot i})$$

Thus, the sequence $Y_1, Y_2, ..., Y_{\lceil n/B \rceil}$ represents the maxima of the blocks. The second, instead, is called **Peak-over-Threshold (PoT)** and it is used to fit GPD and, as the name suggests, it is a simple threshold filter:

$$Y = \{X_i > u\}$$

where u (the parameter that has to be set here) is a predefined threshold. So, the sequence $Y_1, Y_2, ..., Y_m$ represents those measurements that are greater than u, while all the values below the threshold u are discarded. After applying one of the two methods, any traditional parameter estimation algorithm can be used to fit the corresponding distribution (e.g. the Maximum Likelihood Estimator or Probabilistic Weighted Moments).

Anyway, the condition for EVT to be applied is that the sample of execution time observations passes appropriate statistical tests. In that case, we can estimate the probability distribution of the extreme values of the execution time traces of the program.

2.1.3 The EVT hypotheses

Early studies required the sample of observations to be *independent and identically distributed* (i.e. they are mutually independent and each random variable has the same probability distribution as the others). This condition is very difficult to obtain in this context, since a processor with a standard cache would not be able to fulfil this requirement, having subsequent executions of the same task affected by the status of the cache. However, later work by Leadbetter [18] showed that the hypotheses can be relaxed, while maintaining the validity of EVT results if the following properties are proved to be valid:

- **Stationarity**: a sequence of random variables (i.e. a series of observations) is stationary if the joint probability distribution does not change when shifted in time, and hence the mean and variance do not change over time. The most used test to check it is the **KPSS** *(Kwiatkowsky, Phillips, Schmidt, and Shin)* test;

- **Short-term independence**: it requires the elements of the sequence of execution times not to present a statistical dependency with the near other elements. It can be checked through **BDS** *(Brock, Dechert, Scheinkman and LeBaron)* test;

- **Long-term independence**: it requires the job execution time not to present *seasonality*, which means there is no long periodicity. The **Hurst Exponent** *(H)* is the traditional index used to measure

the long-term memory of a time series in financial applications. However, performing a statistical test on H is nontrivial and it does not exist a well-assessed test. Thus, **R/S statistic equation**, which can be directly used as a test, is employed to compute the Hurst index.

Besides the previous hypotheses, other two important hypotheses have to be verified before exploiting EVT for probabilistic real-time:

- **Maximum Domain of Attraction (MDA)**: this requirement does not have a direct correlation with hardware or software features, rather it is a statistical property. From previous statistical works, it is known that most of all continuous distributions satisfy this hypothesis, if the measurements are correctly acquired. During the real measurements of our system, we clearly need to discretize the elapsed time. However, if the time sampling has a sufficient resolution (i.e. sufficiently high frequency of sampling), this hypothesis can be considered valid with a good degree of confidence. Anyway, to check the validity of MDA hypothesis, a Goodness-of-Fit statistical test (GoF) is usually performed a posteriori of the pWCET analysis. This test helps also to identify any error in the estimation phase, because it is able to detect the discrepancies between the estimated pWCET distribution and data that have actually been measured;

- **Representativity**: this is a fundamental issue in applying statistical methods (i.e. MBPTA) to estimate the pWCET distribution of a program. The problem is that the results obtained are only valid for those scenarios of operation for which the sample of observations used in the analysis is representative. So, it is essential that the measurement protocol covers all the future scenarios of operation that could occur in practice. Usually, wide coverage of the different input states, hardware states, and the worst-case path(s) through the program is required. A simple experimental example shows how the estimated pWCET distribution produced depends on the distribution of input values, and hence that if the input data distribution used for analysis does not match the one occurring during the operation of the system, then the results obtained may not be precise. Applying

uniformly distributed input values during analysis may result in poor estimates, as well.

In this thesis, we do not take into account these last two hypotheses, because the former is often true and the latter is dependent on the application code itself. We instead focus on the i.i.d. hypothesis and the three conditions to satisfy when i.i.d. is relaxed.

2.1.4 Application of EVT to pWCET problem

The whole pWCET estimation process can be summarized in the following steps:

1. acquire the time measurements $X_1, X_2, ..., X_n$;
2. check the previously explained hypotheses on inputs;
3. apply BM or PoT filtering to the time measurements to obtain a new set of execution times $Y_1, Y_2, ..., Y_n$ representing the WCET behaviour;
4. run the estimator (e.g. MLE or PWM) to obtain the pWCET distribution;
5. compare the pWCET distribution with the original samples thanks to a GoF test;
6. compute the WCET given the desired violation probability by using the cdf of the estimated pWCET, or vice versa.

The check performed at *point 2* does not always return positive outcomes (i.e. not all the necessary conditions are matched). This could happen due to issues related both to the software or hardware parts of the environment. Regarding the software side, the inputs' correlation is one of the main causes that could affect the applicability of this technique. It happens when the input values depend on previous ones, but also when they are influenced by previous outputs, making the program show some kind of history dependence. The problem may be reduced by changing the way test input is created, but still keeping in mind that the representativity condition must always be satisfied, otherwise the analysis would not be useful to find out what actually happens in the production environment.

Other issues may be given by the structure of the code itself: global variables may store information deriving from previous executions and having an impact on the following observations, plus, different paths in the program may result in different execution time distributions. In this second case, employing per-path approach instead of per-program one could result in an improvement. On the hardware side, instead, the main obstacle for independency conditions to be satisfied is cache content, which may store data that can be re-used in the following executions, determining lower completion times. In this case, proposed solutions are the complete reset of the cache before each execution, or the use of time-randomised hardware (such as random replacement caches), even if it has been proved that the latter is neither sufficient nor necessary condition for the EVT to be applied.

2.2 Statistical tests

A statistical test is a method of statistical inference that aims at verifying a statistical hypothesis, which is a hypothesis that can be verified through the observations of a model. Such observations can be represented by a set of random variables. The first step, when applying this method, is choosing the *Null Hypothesis* (H_0), whose truthfulness is not sure and needs to be checked, and the *Alternative Hypothesis* (H_1), which is the one that can be taken as valid in case the first one is not. The statistical test decides, based on the realization of the observations' random variables, if H_0 should be rejected in favour of H_1, or not. The result of a statistical test is itself a random variable, thus it may produce incorrect results. There might be cases in which it states that the null hypothesis must be rejected even if it is actually true. This is often referred to as *Error of the I type*, and its frequency can be tuned by modifying the *significance level* α of the test, which determines how often this kind of error is going to happen. Typical values for the significance level are 1%, 5%, and 10% and can be selected by the experimenter. Tuning the α parameter also has an impact on another kind of error, called *Error of the II type*, which happens when the alternative hypothesis is wrongly accepted. The frequency of this error increases by decreasing the significance level. After defining the two hypotheses and the α parameter, the next step is choosing an appropriate test suitable for the considered case. Then, observations of the model must

be performed, and from them t_{obs}, the observed value of the statistic T, can be computed. It allows calculating the probability to have an observation at least as extreme as the observed one, which goes under the name of *p-value*. Finally, this value can be compared to the level of significance, to determine the outcome of the test. In particular:

- if $p\text{-}value \geq \alpha$, then there is not enough evidence to be able to reject the null hypothesis H_0;
- if $p\text{-}value < \alpha$, then we can say the null hypothesis is rejected at the chosen level of significance, in favour of the alternative one H_1.

An alternative but equivalent method, the one that will be used in this thesis, splits the possible values of the statistic T into those for which the null hypothesis is rejected (the *critical region*) and those for which it is not. The *critical value*, dividing the two regions, is chosen according to the distribution of the test statistic under the null hypothesis. In particular, the probability of the critical region is equal to α. Whether to accept the null hypothesis H_0 or not is chosen according to the observed statistic t_{obs}:

- if $t_{obs} \notin$ critical region: H_0 cannot be rejected;
- if $t_{obs} \in$ critical region: H_0 is rejected;

2.3 Probabilistic Predictability Index

The different output values of the three previously described statistic tests (Section 2.1.3) do not allow to understand in a clear and immediate way whether the input time series was time predictable or not, nor if EVT can be applied. A unified and thus more quickly readable index providing a numerical value that expresses the fulfilment of the EVT statistical hypotheses has been introduced by Reghenzani et al. [29]. This index is called **Probabilistic Predictability Index** or **PPI**. It is meant to merge the three tests, while maintaining their statistical properties, so that it can be used as a hypothesis test as well. The PPI is defined over the continuous range (0, 1), and PPI values near 0 state there is strong evidence that the time series is not analysable, vice versa for PPI values near 1 the time series is very likely satisfying EVT hypotheses.

2.3.1 PPI construction

In order to merge the three statistics, a common domain D = (0; 1), that is the PPI domain, is defined. Taking into account the desired meaning for PPI and the ranges of the three statistics, the following functions have to be found:

$$f_{KPSS} : (0; +\infty) \to D$$
$$f_{BDS} : (-\infty; +\infty) \to D$$
$$f_{R/S} : (0; +\infty) \to D$$

Under the following constraints:

$$\lim_{x \to +\infty} f_{KPSS} = 0 \qquad \lim_{x \to 0} f_{KPSS} = 1$$
$$\lim_{x \to \pm\infty} f_{BDS} = 0 \qquad \lim_{x \to 0} f_{BDS} = 1$$
$$\lim_{x \to +\infty} f_{R/S} = 0 \qquad \lim_{x \to 0} f_{R/S} = 1$$

Moreover, the rejection property of the test statistics against the critical value has to be maintained. In order to meet this requirement, the f_{KPSS}, f_{BDS}, $f_{R/S}$ transformations have to be continuous, positive, and monotonic functions. The following formulae were proved to satisfy all previous properties:

$$f_{KPSS}(x) = e^{-K_{KPSS} \cdot x}$$
$$f_{BDS}(x) = e^{-K_{BDS} \cdot |x|}$$
$$f_{R/S}(x) = e^{-K_{R/S} \cdot x}$$

Then, K_{KPSS}, K_{BDS}, $K_{R/S}$ have to be selected in order to be compliant with the previous constraints and to get the same critical value for each test. The first value $K_{KPSS} = \frac{1}{4}$ is assigned empirically, then the critical value for KPSS test can be computed as $C^*_{KPSS} = e^{-\frac{1}{4} C_{KPSS}}$. Since the same critical value for the other two tests is desired, their constants have to be computed as follows:

$$k_{BDS} = -\frac{\log C^*_{KPSS}}{|C_{BDS}|}$$
$$k_{R/S} = -\frac{\log C^*_{KPSS}}{C_{R/S}}$$

obtaining:

$$C_{PPI} := C^*_i = f_i(C_i) \quad \forall i \in \{KPSS, BDS, K/S\}$$

Of course, assigning a different value to k_{KPSS} would produce different statistic PPI, for what concerns absolute values. However, the statistical meaning would not be modified, because the critical values would change in a consistent manner. Choosing the value of $k_{KPSS} = \frac{1}{4}$, the obtained C_{PPI} is 0.89 for $\alpha = 0.05$.

Having the three statistics uniformed in the $(0; 1)$ range and with the same critical value, it is finally possible to merge them into a unique index, by following these conservative listed criteria:

- if *all* test statistics are higher than the critical value, the PPI must be higher than the critical value as well;
- if *any* of the three test statistics is lower than the critical value, the PPI must be lower than the critical value;
- if *more than one* test statistics are lower than the critical value, the PPI must be lower than the minimum statistic.

This approach allows us to state that if and only if at least one of the three hypotheses is violated, then PPI will be lower than the critical value and thus the null hypothesis rejected. So, the following transformation is applied:

$$PPI := \begin{cases} \min\limits_{\forall i} f_i(S_i) \cdot \prod\limits_{i \in v^*} [1 - (C_{PPI} - f_i(S_i))] & v \neq \varnothing \\ \dfrac{1}{3} \sum\limits_{\forall i} f_i(S_i) & v = \varnothing \end{cases}$$

where $v = \{i \,|\, f_i(S_i) < C_{PPI}\}$ and $v^* = \{v \setminus \arg\min_{\forall i} f_i(S_i)\}$ are respectively the *violation set* and the violation set, excluding the minimum.

So, if no violation occurs in the three tests, PPI is computed as the average of the three values (which means they all have the same weight), which is greater than C_{PPI}. Otherwise, the PPI is equal to the minimum statistic potentially multiplied by other statistics that violate C_{PPI}, leading to a PPI value lower than C_{PPI}. Summarizing, it is possible to compute the value of PPI through the previous equation and then the following null hypothesis (H_0) has to be rejected in favour of the alternative one (H_1) when $PPI < C_{PPI}$. The statistical testing hypotheses scheme is then:

H_0: the time trace verifies the EVT hypotheses

H_1: at least one EVT hypothesis is violated

CHAPTER 3

State of the Art

While studies with traditional techniques on WCET have been researched for a long time, the branch of probabilistic timing analyses began to spread only in recent years. In this chapter we go through its development, starting from the first scientific work, and showing the evolution of the estimation procedure, both in terms of how to verify the EVT hypotheses, and how to perform the statistical tests to check them. After that, we move to the description of frequently used benchmark suites, which are very important since they form a common basis from which we start our analysis, and allow us to make comparisons between the results of different methods.

3.1 Measurement Based Probabilistic Timing Analyses

The use of EVT to estimate the maxima of a distribution of a series of execution times of a program was introduced in 2000 by Burns and Edgar [4]. The reason was the increasing complexity of hardware architectures that makes more and more complex and computationally expensive the estimation of the WCET using the traditional techniques, as we also

described in Section 1.2.1. The following year, the statistical estimation of the pWCET distribution was proposed by Edgar and Burns [7]. This work was proved to contain some issues: first of all, they were using a Gumbel distribution to directly fit the observations, instead, it should be used on maxima of subsets of execution times, as pointed out by Hansen et al. [17]. Furthermore, χ-squared test is used to fit the scale and location parameters of the Gumbel distribution, but this test is not meant for this purpose, as underlined by Cucu-Grosjean et al. [5]. These issues were solved in 2009 by Hansen et al. [17], by applying BM method (see Section 2.1.2), starting from a block size of 100 and tuning the parameters until the χ-squared test proved sufficiently high goodness of fit. This method provided much better results than taking the highest measured execution time as WCET, however, some cases in which the estimation was too optimistic occurred. In 2010, other issues were found out to be solved in order to have better estimation by Griffin and Burns [14]. These included the fact that Gumbel distribution (which is continuous), was used on execution times that may had large discrete steps, and also the requirement of the measurements to be statistically independent and identically distributed (i.i.d.). The independence assumption can be broken by factors such as caches (storing information from previous executions that can be re-used and allows to save some time), or input which is dependent on the previous output (and thus creating some kind of historical dependence in the trace), while different paths in the program may determine different behaviour of the program itself, this causing issues with the identical distribution constraint. In 2012, Cucu-Grosjean et al. [5] introduced a new method for applying EVT to estimate pWCET, by exploiting end-to-end execution time measurements and involving three different kinds of tests: two sample Kolmogorov-Simirnov test, and Runs test to verify i.i.d. hypothesis and Exponential Tail test to check whether the distribution of the maxima (obtained through BM algorithm even in this case) fits Gumbel distribution. Another key point discussed in this paper is how to keep i.i.d. requirement valid even in multi-path programs. Two solutions were proposed: the first is to randomize input selection and keep the sequential order of the measurements, the second is to test all the possible inputs and then perform random sampling without replacement when dividing them into blocks.

From this point on, in the literature, the focus was trying to relax the i.i.d. hypothesis, thus making the cases in which observations show dependencies between themselves easier to analyse. Such cases are very likely to happen. In this regard, the first work was by Santinelli et al. [34] in 2014, where weaker hypotheses are checked, such as stationarity, verified by considering autocorrelation through lag plots, and extremal dependence, which is studied with extremograms, that show whether the extremes are clustered or distributed throughout the whole time series. Furthermore, the effects of changing the size of the blocks in BM procedure and the threshold in PoT are investigated, finding out high sensitivity to these two parameters and performance degradation, when block size or the threshold is increased too much. In the same year, Berezovskyi et al. [3] provide a critic about the Runs test previously used by Cucu-Grosjean et al. [5], suggesting to replace it with tests based on auto-regression and autocorrelation. Lag plots (for autocorrelation, together with Ljung-Box test) and extremograms (for extreme values dependence) are still used; in addition, autocorrelation tests combined with the notion of stationarity are performed to verify statistical independence, and stationarity is determined by an autoregressive model. In 2016, Guet et al. [15] introduced a logical work-flow that requires the verification of the following properties: (I) stationarity, (II) short-range dependence, (III) local independence of the peaks, and (IV) that the empirical peaks over the threshold follow a GPD. Kwiatowski-Philips-Schmit-Shin (KPSS) test is used to determine the stationarity, Brock-Dechert-Scheinkman (BDS) test to evaluate the short-range dependence, and Extreme Index is used to determine whether the peaks can be considered independent.

Another crucial point is representativity, i.e. the obtained results are valid only for those scenarios which are represented by the input used for the analysis. In 2016, Lima et al. [22] explained that since the peaks' distribution is highly related to how often each input occurs, there may be more than one distribution describing the behaviour of the program; thus, they introduced the concept of weak pWCET, which is related only to the distribution of the input used for the analysis. Furthermore, it was also pointed out that GEV distribution should be used to fit data, instead of a more specific one (e.g. Gumbel), and various examples in which peaks belong to all the three classes (reversed Weibull, Gumbel, and Frechet), to-

gether with cases in which none of them can be used are provided. Finally, the authors presented evidence that time randomization is not necessary, nor sufficient for the EVT to be applied, similarly to what is claimed by Lima and Bate [12]. In this paper, the authors introduced Indirect Estimation in Statistical Time Analysis (IESTA) method, in which a random component (e.g. a value taken from a normal distribution) is added to the time measurements, increasing the probability that EVT can be applied without introducing too much pessimism in pWCET estimation. Issue of representativity, together with the one of reproducibility, is also discussed by Maxim et al. [24]. For what concerns representativity, they claimed there must exist a certain value k for which taking any number of observations greater than this value guarantees that the obtained distribution would finally converge to a sufficiently close approximation of the actual pWCET. Instead, reproducibility means that two different traces obtained by starting from the same initial conditions will lead to the same pWCET distribution, or at least to two close ones. In 2017, Santinelli and Guo [33] proposed a new framework, in which each task is represented by a collection of pWCETs (each one related to a different input distribution) and not by just a single one. Enumerating all the possible environmental conditions is a complex problem, anyway, some of them can be eliminated due to the dominance relationship, while incomparable ones can be summarized by a single pWCET upper-bounding all of them. The authors suggested using this kind of approach on mixed-criticality systems. In the same year, Abella et al. [2] introduced a new way to select the parameters of PoT and BM algorithms, so that enough values from the actual tail of the distribution are passed to EVT, including only a few of them from the non-tail part of the distribution. The method is based on the assumption that real-time systems have finite execution times, so heavy-tailed distributions can be excluded, and exponential-tailed distributions (i.e. Gumbel) can be used as upper bound of the light-tailed ones, since it is acceptable to have slightly pessimistic pWCET, while an optimistic one is not allowed. So, plots showing how the value of the CV (coefficient of variation, i.e. standard deviation over mean) changes according to the different tuning parameters are employed, with the aim of finding a configuration in which the exponential tail hypothesis cannot be rejected, and the value of the CV is close to 1 (also needed to have an exponential distribution). This

method was, however, proved to be optimistic, and consequently unsafe, under certain circumstances [27].

Up to this point, most of the research papers assumed EVT hypotheses as always satisfied, or at most based their verification on strategies based on expert knowledge, that cannot guarantee high levels of accuracy. In this regard, recent researches, such as [30] and [29], have focused on the usage of statistical tests to correctly verify the hypotheses. In these works, PPI, an index showing at which level hypotheses are satisfied in terms of numerical values (and thus in a more objective way), was introduced. Of course, also statistical tests are subject to errors, this is why in [31] their degree of confidence was studied, with particular focus on the minimum number of input samples (that until then was usually empirically determined) required to reach sufficiently high level of significance. Furthermore, in [32], the relationship between statistical tests and the estimators' accuracy was studied, and techniques to study and cope with uncertainties were proposed.

Embedded systems is not the only field in which pWCET can be employed. In this recent article [11], its applicability to high-performance computing (HPC) was inspected. Authors explained than in this case there are further issues that must be faced, since considered systems have higher complexity (thus it is difficult to have full coverage of execution conditions guaranteed), and multi-threaded applications are run on them. However, they prove how techniques based on MBPTA and software randomization can provide tight pWCET estimates. Moreover, European *RECIPE* project [10] is aiming at improving runtime resource management techniques, to increase predictability in terms of reliability and timing, while ensuring full exploitation of available resources.

3.2 WCET benchmarks

Finding the WCET of real-time software is crucial when developing and verifying real-time systems. These bounds must be safe and, preferably, tight (i.e. as close to the actual WCET as possible) and WCET analysis attempts to deliver such a bound. Several WCET analysis tools have emerged in recent years and, of course, it would be very important to find out which of them provide the best results. However, a comparison between these tools, and the associated methods and algorithms, requires

common sets of benchmarks. The typical evaluation metric is the accuracy of the WCET estimate, but other properties such as performance (i.e. scalability of the approach) and general applicability (i.e. ability to handle all code constructs found in real-time systems) also have great importance. In summary, it is very useful to have easily available, deeply tested, and well documented common sets of benchmarks in order to enable comparative evaluations of the different existing tools. Open source benchmarks are even more attractive, not only because they improve the reproducibility of the experiments and the possible exploitation by other researchers, but also because they allow any person to check, study, and modify the source code.

3.2.1 Mälardalen WCET

The Mälardalen WCET benchmarks [16] were collected in 2005 from several types of research and industrial use-cases within the WCET field and are all available on a web page[1]. They have been extensively used, mainly for evaluation of WCET algorithms and tools in research papers, and for comparisons between WCET tools. The benchmarks have been used as test programs also for other purposes, like dynamic programming, migration of real-time tasks, and scratchpad memory management.

The researchers marked each benchmark program with some properties, indicating whether it is a single path program, contains (nested) loops, uses arrays or matrices, or includes recursion and other characteristics. The web page also includes additional information for the benchmarks:

- *input limits*: each benchmark contains its own input, so the programs in the benchmark can all be run "as is", that means that they execute a single path. However, most realistic programs are run with different inputs at different invocations, since if the inputs can affect the control flow, the program's WCET is usually highly dependent on inputs. Thus, feasible intervals for input variables are given, sometimes together with information about the best and worst cases;
- *loop bounds*, i.e. the number of times each loop contained in the benchmarks may be traversed. The loop bounds for each program are either exact (in case the program has been run only with the input

[1] URL: `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

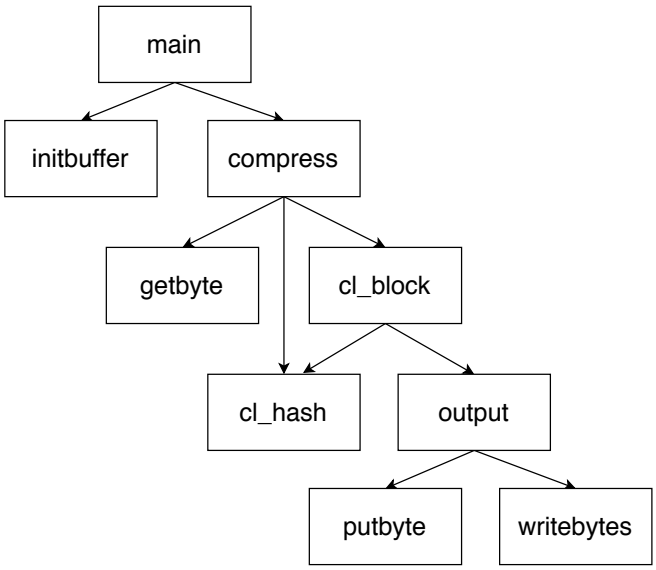


Figure 3.1: *An example of function call graph, taken from benchmark "compress".*

in the code) or the maximum possible with the feasible inputs, as previously defined;

- *call graphs* and *scope hierarchy graphs*, generated by the SWEET tool and provided as PDF files, useful to easily understand function calls and loop invocations in the programs. The root scope is typically the main function, or the top function in a subgraph, while the other scopes are either functions or loops, and constitute possibly looping entities in the programs. The arrow from one scope to another below represents a call in the case of a function scope, or a loop invocation in the case of a loop scope. An example of graph, taken from benchmark *compress*, is shown in figure 3.1.

There are also some drawbacks to underline. Being them not industrial real-time applications, the benchmarks are mostly small programs (all except two are less than 900 lines of code). The WCET benchmarks typically test just a few specific programming constructs, and, therefore, they cannot test how algorithms and tools scale with larger programs. Furthermore, the whole program often fits in a cache, making the evaluation of cache analyses hard. Moreover, benchmarks are mainly focused on flow analysis, without testing hardware features like instruction caches, data caches, or branch predictions, and some program constructs such as complex

low-level code (e.g. bit-operations and shifts), use of dynamic memory, mode-specific behaviour, tasks with multiple roots, tasks wrapped in a loop, and programs using function pointers are missing. There is also weak support for measurement-based WCET analyses:

- the inputs of each program are fixed in the file and therefore different inputs cannot be supplied as parameters;
- the worst-case test vector is not given;
- a common set of realistic test vectors is missing, thus different tools and techniques are very likely to generate different inputs, making comparison awkward.

Finally, these benchmarks are not suitable for parallel computing, which would be important since WCET analysis has moved towards multi-core systems.

3.2.2 TACLeBench

TACLeBench [8] provides a freely available and comprehensive suite of 53 benchmark programs for timing analysis and related research topics. They have already been widely used in different study fields, such as compiler optimizations, measurement-based and hybrid analyses of WCET and hardware design. The benchmarks included in TACLeBench are provided as ISO C99 source codes, and are totally self-contained, that means no dependencies to system-specific header files via #include directives or an operating system exist, but all input data is part of the benchmark source code. Almost all benchmarks are processor-independent, i.e. they can be compiled and evaluated for any kind of target processor, and any update of TACLeBench is carefully managed with snapshots and versioning, so that it is clear which code has been used in a research experiment. All benchmarks are completely annotated with flow facts, distinguishing between so-called flow restrictions[2], loop bounds, and entry points. These flow facts are directly incorporated into the C source codes using pragmas. The benchmarks are divided into five classes:

- *kernel benchmarks*, synthetic benchmarks implementing small kernel functions;

[2]Indications about feasible paths

- *sequential benchmarks*, implementing large function blocks, such as encoders and decoders, which are used in many embedded systems and covering graph search, cryptographic algorithms, compression algorithms, etc.;
- *artificial test benchmarks*, that can be used to stress test WCET analysis tools;
- *application benchmarks*, derived from real applications (such as a lift controller and a controller for an electric window in a car) and provided with simulated input stimuli;
- *parallel benchmarks*.

The original versions of the benchmarks have been rewritten in order to split the functions of input data initialization, and the benchmark itself. The TACLeBench suite includes all input data hard-coded into the C source, which unfortunately turns them into single-path programs. Another consequence of the fixed input data, and also of the fact that some programs did not provide any return value, was that compilers with optimizations turned on could optimize most of the code away. To overcome this issue, with respect to the first version of the suite, the way input data is represented in variables has been changed (making them volatile), and the return of the main function has been made dependent on the benchmark calculation. Also, some benchmarks contain target-dependent code and a few benchmarks are byte-order dependent, when there is no standard way in C to detect the byte order of a processor. Finally, some benchmarks can be executed only once, because either they rely on global initialization, or they improperly use the dynamic memory allocation.

3.2.3 MiBench

MiBench [23] is a suite of 35 benchmarks, that presents similarities to EEMBC suite[3], but with the difference that source code is freely available. The programs are split into 6 different categories, ranging through the whole embedded domain: Automotive and Industrial Control, Network, Security, Consumer Devices, Office Automation, and Telecommunications. All benchmarks are written in standard C, which makes MiBench portable

[3]Suite of benchmarks created by Embedded Microprocessor Benchmark Consortium

to any platform with compiler support. Each program is provided, when needed, with examples of input parameters for both light-weight and large analyses, allowing to perform tests respectively in a simple and in a more stressful way, computationally speaking. Furthermore, statistics and graphs regarding the length of the benchmarks in terms of lines of code, cache misses, branch misprediction rates, and the distribution of the different classes of instructions (branches, integer, floating-point, and memory) in the different categories and single programs are provided.

CHAPTER 4

Methodology and Experimental Framework

In this chapter, the adopted methodology for the subsequent experiments and analyses is described. We start by focusing on the software part, explaining how already existing benchmarks have been modified in order to make them all follow the same general structure. After that we move to the hardware side, describing what instruments have been used for the analyses, and how they have been configured. All paragraphs contain some pieces of pseudo-code, providing a quick idea about the interactions between the different components, and showing the operation of the considered programs.

4.1 Overview of the experiments

The objective of our experiments is to check whether considered benchmarks are time predictable, i.e. if execution time series obtained by running them multiple times verify EVT hypotheses. To do this, we ex-

ploit PPI, which, as explained in Section 2.3, provides a numerical value stating whether the hypotheses of the three statistical tests are valid or not. Furthermore, we fit data on GEVD and GPD, with particular focus on the shape parameter ε, providing information about the tails of the distributions.

Checked programs come from the most common benchmark suites, but we also create some new ones by modifying them, and we write others from scratch, as well. Histograms showing the distributions of the execution times of all the tested benchmarks are available in Appendix B. All the programs are tested on different hardware architectures and with different configurations. First, they are executed on a microcontroller by STM32, and two different methods of measuring time are employed: the internal timer of the board, and a logic analyser. Then, benchmarks are run on a Raspberry PI 4, with two different setups: in fact, besides executing them on the standard Linux version, experiments are also repeated after the application of the PREEMPT_RT patch.

4.2 Benchmark preparation

The initial step was collecting a set of benchmarks to be used for the analyses. Most of them were taken from the *Mälardalen benchmark suite* [16], but some of them come from different sources, such as *MiBench* [23] and *MediaBench* [20], as previously presented in Section 3.2. First of all, all benchmarks have been converted to the standard C format, eliminating unnecessarily complicated constructs or deprecated syntax from the early C standards (e.g. the function parameters expressed with the K&R style). After that, all *main* functions have been removed, substituted by functions with the same name of the benchmark itself (i.e. name of the *.c* file). In case the program contained a method with the desired name already, it has required rename as well.

4.2.1 Input generation

Most of the benchmarks contained hard-coded input values, which did not allow us to test different input sizes and values, necessary when a WCET analysis is performed. Thus, predefined initialization has been replaced by a custom function, designed to assign random values to all needed variables

and arrays, thanks to a Pseudo-Random Number Generator (PRNG). The function makes use of a *seed*, exploited for the generation of the random numbers. This allows us to have a different initialization by just changing the value of the seed at each execution. Having a configurable seed makes experiments replicable, since feeding the random function with the same seed means generating the same random series. The employed random function was *lfsr113* [19], a 32-bit PRNG returning numbers in the interval [0,1), and guaranteeing a period $\rho = 2^{113}$. Previously, another PNRG, taken from *Mälardalen* benchmark suite, had been tested, however, its frequency proved to be too low, affecting the results of the experiments by introducing dependencies. Another option was the usage of *rand()* function, but it was excluded to keep the benchmarks independent from C libraries.

4.2.2 Execution time measurements

In addition to the seed, we added as input parameters to all the considered benchmarks two function pointers, which are used for the computation of the execution times of the benchmarks. Algorithm 1 shows the general structure of a benchmark. The function at *line 4* corresponds to the call of the first pointer at the beginning of the execution, right after the initialization of the variables (in order to avoid measuring the variability that may be present in the previously described random function to affect the results). The second function pointer is called at the end of the computation (*line 6*), just before the *return* instruction. These two pointers can be provided by the experimenter depending on the platform he or she uses to perform the tests. For example, in a Linux system, the two functions can be implemented with *clock_gettime* primitives, and the computed difference between the two values represents the execution time.

4.3 STM32 Nucleo Board

4.3.1 Hardware architecture

The first set of experiments has been carried out using a *STM32L010RB microcontroller*, manufactured by *ST Microelectronics*. The board is equipped with *Arm Cortex-M0+* 32-bit RISC core operating at 32 MHz and embeds high-speed memories: 128 Kbytes of Flash program memory, 512

Algorithm 1: Benchmarks' general structure

```
1  Function benchmark(seed, start_function, end_function):
2      PRNG_setseed(seed);
3      foreach var ∈ variables_to_initialize do
4          var := PRNG_random();
5      end
6      start_function();
7      execute_benchmark(variables_to_initialize);
8      end_function();
9      return;
10 end
```

bytes of data EEPROM, and 20 Kbytes of RAM. Several communication interfaces are also present: one I2C, one SPI, one USART, and a low-power UART (LPUART). Finally, 51 *General Purpose Input/Output* (GPIO) pins are available. Figure 4.1 shows the block scheme of the board.

4.3.2 Code structure

The base code including the setup of the board, the initialization of some of the GPIOs, the clock rate, and other settings, has been automatically generated by using the dedicated software *STM32CubeMX*. These parameters have then been edited and enriched according to the purposes of each experiment. Two different versions of the code have been written, one for each method employed to measure the execution time. The reasons why we tested two different measurement methods will be explained later in Chapter 5. The first and simpler way to perform measurements was by exploiting the internal timer of the board. A sketch of the code used in this case is shown in Algorithm 2. The *wait* function at *line 1* is called to let the microcontroller complete its initialization and the clock to stabilize, otherwise, the first measurements may result incorrect. At *line 2*, the random function is initialized with a certain seed, while at *line 4* the function of the corresponding benchmark is called, and the two parameters are the two pointers to the functions called respectively at the beginning and at the end of the computation. In this case, their behaviour is just to store into variables *start* and *stop* the time instant (i.e. amount of time elapsed since

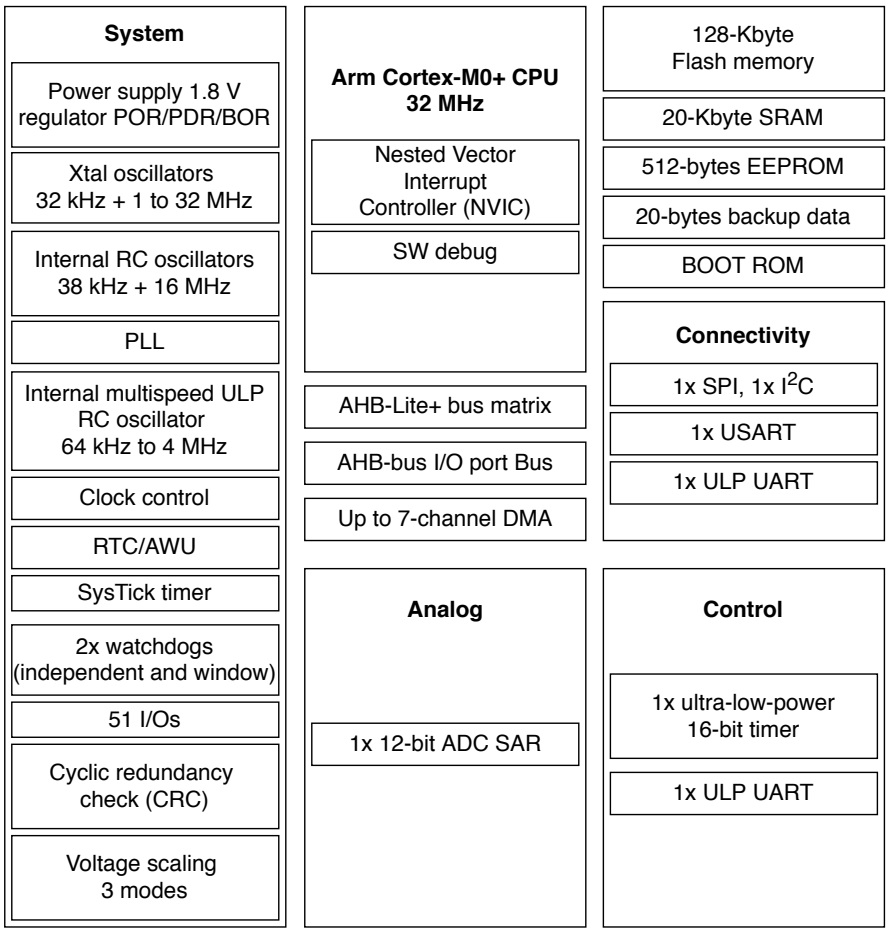


Figure 4.1: *The block scheme of STM32L010RB microcontroller.*

the beginning of the computation, in terms of microseconds) in which they are called. The total execution time can then be derived by simply subtracting the starting time from the end one. However, the time counter has a limit (65536), after which it is reset to 0, and it may happen it overflows in the middle of the benchmark, thus bringing to negative execution time. We implemented a function named *correctOverflow* in charge of fixing the issue, by checking whether the value of time is lower than 0, and simply adding 65536 to it in case it is. Finally, the measured time is sent to the virtual serial port, and can be read from the host computer the board is connected to.

The other method employed for the time measurements makes use of an external logic analyser, which is connected to one of the GPIO pins of the STM32 board, whose function is to signal when each execution of

Algorithm 2: STM32 Nucleo board code – Internal Timer

```
1 wait();
2 random_seed := SEED;
3 foreach i ∈ 1 .. Num_Of_Mesurements do
4     execute_benchmark(begin, end);
5     time := stop - start;
6     correctOverflow(time);
7     send(time);
8 end
```

the benchmark starts and stops. The behaviour is the following: the logic analyser monitors the status of the pin it is connected to, waiting until the signal is kept low and triggering as soon as a rising edge is detected. In fact, this means that the first execution of the benchmark has started on the board. From now on, the analyser keeps sampling the value (i.e. high or low) of the pin at the frequency of *100 MHz*[1], and each time a falling or rising edge is detected, the timestamp (i.e. the moment in which the event occurred) is saved. Given the sampling frequency, the precision of the performed measurements is 10 ns. At the end of the last execution, all stored timestamps are returned by the software of the logic analyser in a *.csv* file, which must then be correctly parsed in order to obtain the actual execution times of the benchmarks. The structure of the codes of both the board and the logic analyser is presented in Algorithm 3 and Algorithm 4. At the beginning of Algorithm 3 (showing the code of the board), the waiting time has been removed since the internal timer is not responsible for the measurements anymore, so there is no need to wait until its initialization is completed, while the seed of the random function is still set and its value is the same that had been used in previous measurements, in order to have the same input values and thus comparable results. Unlike before, in this case, *begin* and *end* functions just change the state of the output pin (from low to high and vice versa) used for the communication with the logic analyser. Furthermore, the instructions related to the management of the final result have been removed, because in this case it is the logic analyser in charge of computing the actual

[1]The frequency 100 MHz was the highest possible on the instrument available during the experiments.

Algorithm 3: STM32 Nucleo board code – Logic analyser measurements

```
1 random_seed := SEED;
2 foreach i ∈ 1 .. Num_Of_Mesurements do
3 |   execute_benchmark(begin, end);
4 end
```

execution time.

Algorithm 4 shows the code related to the logic analyser. The behaviour of the first part (*lines 1-7*) is as previously explained: the analyser waits for the first rising edge and then keeps sampling and storing the desired timestamps until the last execution is over. After that (*lines 8-11*), vector *times* is built by computing the difference of consecutive elements of vector *timestamps*. Thus, *times* contains both the desired execution times (i.e. difference between a rising and a falling edge) and the idle times between the end of an execution and the beginning of the following one (i.e. difference between a falling and a rising edge), which are not interesting for the analysis and must be discarded. This is done in the last part (*lines 13-17*) of the previous code, where all the values in odd positions of vector *times* are inserted in vector *executionTimes*, while the others are eliminated.

4.3.3 Setup for Real Applications

An issue that had to be faced when dealing with the *STM32 Nucleo* board was the limited size of the data memory *SRAM*. In a few cases, in fact, the size of the arrays and matrixes included in the original benchmarks had to be slightly reduced in order to have everything fitting the available space. Furthermore, while in most of the cases benchmarks are fed with random input, generated at runtime during the execution of the program itself, in other cases it was not possible. For example, real applications (such as *TCAS*, presented in Section 6.5, or *audio compression* benchmark, presented in Section 6.4) required realistic inputs to be provided. Neither hard-coding this data into the source *.c* file was a viable option: putting the whole values needed to perform all necessary executions (about 10.000 for each benchmark) would have meant to go beyond the limits of the

Algorithm 4: Logic analyser code

```
1  timestamps := [];
2  foreach i ∈ 1 .. Num_Of_Mesurements do
3  |   wait_RisingEdge();
4  |   timestamps.add(current_time);
5  |   wait_FallingEdge();
6  |   timestamps.add(current_time);
7  end
8  times := [];
9  foreach i ∈ 1 .. timestamps.length do
10 |   times.add(timestamps[i+1] - timestamps[i]);
11 end
12 executionTimes := [];
13 foreach i ∈ 1 .. times.length do
14 |   if i is odd then
15 |   |   executionTimes.add(timestamps[i]);
16 |   end
17 end
```

available SRAM and flash regions. Changing the hard-coded input at each execution was clearly not a solution, because it would have required to keep manually modifying the code to reach the desired number of executions. That is why both *.csv* and *.txt* files containing realistic input values for the different benchmarks were employed. These kinds of files cannot be accessed and read directly from the board itself, therefore we implemented a data exchange mechanism through serial communication between the board and the host computer. The latter was in charge of reading the previously mentioned files containing the input and sending data to the board for the execution. In Algorithm 5 and Algorithm 6 it is possible to see respectively the script on the board and the host side.

At *line 1* of Algorithm 5, the number of chars to be received at each execution is set. In fact, function reading characters from the serial port used at *line 4* needs to know in advance how many it has to receive before storing them into variable *data*, which is a string. Thus, it must be turned into a numerical format to let the function executing the actual benchmark be able to use it. The function *parse* is called at *line 5*, and is responsible for

Algorithm 5: Board receiving applications' realistic data code

```
1 numOfChars := CHARS;
2 input := [];
3 for i in 1..numOfExecutions do
4 |   data := read_n_chars(numOfChars);
5 |   input := parse(data);
6 |   start();
7 |   execute_Benchmark(input);
8 |   stop();
9 end
```

splitting the string into the tokens it is composed by (the comma is usually used as separator character in .csv files), and for converting substrings into numerical values, that can be then stored into *input* array. Of course, the whole initialization part is excluded from the time measurement, which only includes the actual benchmark execution.

Regarding the host side (Algorithm 6), at the beginning (*lines 1-2*), both the number of characters to be sent each time and the amount of time the host needs to wait before sending the next chunk of data are set. The former is needed to send at each execution the amount of data the board is expecting. Sending less would make the board wait for the remaining part, and that would be taken from data related to the following execution, which is clearly undesired behaviour. The latter, instead, must be tuned according to the execution time of the benchmark: a too short delay would cause data to be sent while the board is still performing the previous execution and is not ready to receive the new stream, yet. After that, at *line 3* the host opens the file containing the input and keeps repeating these operations until the last line is reached: at the beginning, it reads the first line and stores it into *line* variable (*line 4*); then, after computing how many characters the string consists of (*line 6*), some x characters are added at the end of it to reach the amount of data the board is waiting for (*lines 7-10*); finally (*lines 11-13*), the board sends the data, reads the following line of the file, and stops for the previously set amount of time. The process is repeated until the new line is empty. In that case, the file is closed and the process stops.

Algorithm 6: Host sending applications' realistic data code

```
1  numOfChars := CHARS;
2  delay := DELAY;
3  data := open(input_file);
4  line := readLine(data);
5  while data != NULL do
6      chars := length(line);
7      while chars < numOfChars do
8          line.append('x');
9          chars++;
10     end
11     send(line);
12     line := readLine(data);
13     wait(DELAY);
14 end
15 close(data);
```

4.4 Raspberry PI

Besides in a time deterministic environment as in the previous case, benchmarks have also been tested on a system where some randomness is involved. For this purpose, a *Raspberry Pi 4* has been employed.

4.4.1 Hardware architecture

The board we used is equipped with quad-core *Cortex-A72* 64-bit CPU, running at 1.5 GHz, and 4 GB *LPDDR4-3200 SDRAM*. Besides 40 GPIOs, it also hosts 4 USB ports (2 with 3.0 and 2 with 2.0 specification), 2 micro-HDMI ports and one SD-slot (used for loading the OS and for data storage). Supported connection are *IEEE 802.11ac* wireless (2.4 GHz and 5.0 GHz), Bluetooth 5.0, and Gigabit Ethernet.

4.4.2 Software configuration

Two different kinds of configuration have been tested:

1. First, programs have been run on the standard command-line Linux version *Raspberry Pi OS 1.4*;

Algorithm 7: Raspberry PI 4 measurements code

```
1  struct start, stop;
2  time := [];
3  random_seed := SEED;
4  foreach i ∈ 1 .. Num_Of_Mesurements do
5  |   execute_Benchmark(begin, end);
6  |   time[i] := (stop.sec - start.sec)*10^9 + (stop.nanosec - start.nanosec);
7  end
8  Function begin():
9  |   clock_gettime(CLOCK_MONOTONIC, &start);
10 |   return;
11 end
12 Function end():
13 |   clock_gettime(CLOCK_MONOTONIC, &stop);
14 |   return;
15 end
```

2. Then, the *PREEMPT_RT Patch* has been applied, allowing almost all of the kernel code (with the exception of a few very small critical regions) to be preempted. This increases the time predictability, but it reduces the overall system throughput. Approaches to use this patch for building real-time Linux-based systems are described in detail in [28].

Furthermore, all benchmarks have been forced to be executed on one of the 4 available cores. Most of the interrupts have been moved from the core where the task is executed (some of them that could not be moved), to reduce even more the interferences. The code used in this case is quite similar to the one for the measurements with the internal timer of the *STM32 Board*, but it still has some differences, as shown in Algorithm 7.

The seed of the function generating random numbers is set to the same values of the STM32 experiments, so that the same input values have been generated as in the previous cases. Furthermore, two structs are defined to store the results of the time measurements, which are performed through the native *clock_gettime* function, which returns the current time by means of two different values: using *CLOCK_MONOTONIC* as an

input parameter to the function, *sec* indicates the number of seconds since some arbitrary fixed point in the past, while *nanosec* stores the number of nanoseconds expired in the current second. This second value increases by some multiple of nanoseconds, according to the system clock's resolution. To calculate the actual execution times, the difference in terms of seconds between the end and the beginning of the benchmarks is first converted into nanoseconds (by multiplying it with one billion), and then added to the difference in terms of nanoseconds. An alternative method that could have been used, similarly to the one of STM32, is to employ one of the GPIO pins of the *Raspberry PI* as *start/stop* signal, and then measure absolute time through a logic analyser.

CHAPTER 5

Experiments on the STM32 Board

This chapter presents the results of the experiments involving already existing benchmarks and their modified versions (i.e. programs in which the sizes of variables or the number of iterations of cycles are not fixed anymore, but sampled from different kinds of distributions). Outcomes have been obtained by measuring time with both STM32 board internal timer, and through a logic analyser, and comparisons between the two different methods are made. In particular, the goal is to check whether the two methods always lead to the same output, or if there is any difference between the two. We expect the timing stability (i.e. measurements with the internal timer of the board) to produce better results in terms of PPI satisfaction. However, a small degree of variability (given by the noise introduced from the logic analyser) might be also positive in case benchmarks are showing a lack of variability, as already stated by Lima et al. in [21]. Therefore, an experimental evaluation is needed to verify the satisfaction of EVT hypotheses in the two cases.

5.1 Original benchmarks

5.1.1 Experimental setup

The first experiments have been executed employing the *STM32 Nucleo Board*, described in Section 4.3, and involving only the already existing benchmarks, with no changes but the ones already described in Section 4.2: code has been re-written in compliance with the standard C format, programs containing hard-coded input have been changed so that it could randomly be generated through a random function fed with a seed, and two functions (as well as related function pointers) have been inserted after the initialization part and at the end of each benchmark, to make the execution time measurements possible. The set of programs considered at this stage is made of 34 benchmarks: 29 are from the Mälardalen suite (almost all the benchmarks belonging to this suite have been considered in this work, with the exception of the ones resulting in very high execution times and the ones for which determining the feasible input region was too difficult), 4 are from MiBench, and 1 was taken from the *WCET Tool Challenge* held in 2014.

5.1.2 Methodology of the experiments

As previously described in Section 4.3.2, two different measurement techniques were used to gather the execution time values: first, time was directly measured through the internal timer of the board itself, then, measures were repeated by using a logic analyser. The benchmarks use a GPIO to expose the execution time: the logical analyser calculates the time between the rising edge (i.e. start of the execution) and the falling edge (signalling the end of the execution) of the GPIO pin. With the first method, results were printed from the board on the serial port, through which they could be read by the host and turned into a *.txt* file. In the second, instead, it was the software of the logic analyser to be in charge of building the final *.csv* file. In both cases, the files were then parsed with *MATLAB* (one of the most common software for scientific computing), in order to perform three different statistical analyses: *PPI* (probabilistic predictability index), *gevfit* (returning the maximum likelihood estimates of the parameters for the generalized extreme value distribution), and *gpfit*

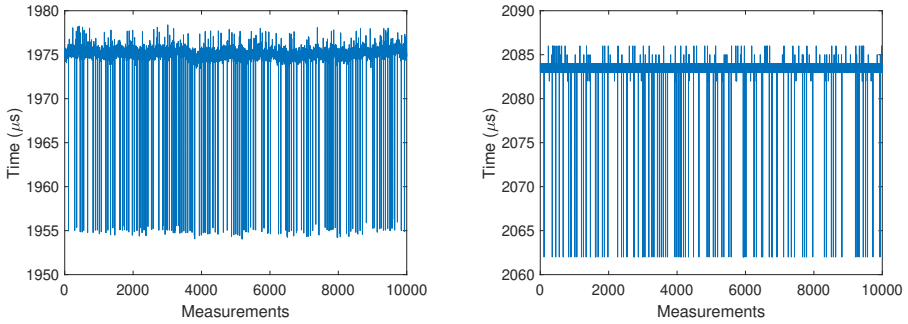
(similar to the previous one, but for generalized pareto distribution). With PPI we aimed at checking if the EVT hypotheses were satisfied, while with gevfit and gpfit we computed the distribution parameters and, in particular, the ε parameter, which provides an indication about the shape of the tail of the distribution, as already explained in Section 2.1.2.

5.1.3 Results

Table 5.1 and Table 5.2 contain the results obtained by the execution of all the benchmarks on the *STM32 Nucleo board*. In Table 5.1, the time has been measured exploiting the internal timer of the board itself, while Table 5.2 refers to the measurements made through a logic analyser. The first column contains the computed value for the *PPI test*, whose hypotheses are respected when the index is greater than the critical value, which, as written in the second column, is always 0.891, computed for a value of α of 0.05, as explained in Section 2.3.1. The following three columns store the single results of the three statistical test PPI test is composed by, while in the last two it is possible to find the values of the shape parameter ε, respectively for the generalized extreme value and the pareto distributions. Results show that in 13 out of the 34 considered benchmarks (i.e. about the 38% of the performed experiments) PPI hypotheses are satisfied for both results obtained from the logic analyser and the internal timer, while in 17/34 (50% of the cases) at least one of the necessary conditions is not valid for both the instruments. Finally, there are 4 benchmarks for which the internal timer and the logic analyser lead to different conclusions. There were also some cases among the previous ones in which, although both methods were giving negative output, the values of PPI were heavily different. In all these cases, it is the value of the index derived from the internal timer's measurements to be higher than the logic analyser's one, while it never happens the opposite. This is the case for instance of benchmarks *crc*, *matmult*, and *shadriver*.

5.1.4 Discussion

Through deeper analysis of previously mentioned cases (in particular by looking at the plot of the time series), it is clear that very different values of PPI between the two methods are obtained when there are just small differences between the measured times from execution to execution.



(a) Logic analyser **(b)** Internal timer

Figure 5.1: *Plots of time series from "fdct" benchmark. It is possible to notice the random noise in the logical analyses plot (a) with respect to the internal timer (b).*

As one would expect, the internal timer measures execution time very precisely (but it does not consider any possible clock skew), while the usage of the logic analyser, that has to deal with GPIOs and electrical signals, introduces some noise, which is responsible for the previously explained behaviour in this specific situation. This is clearly visible in Figure 5.1, showing the plots of the time series obtained by executing *fdct* benchmark.

5.1.5 Final considerations

From the previous results, it is possible to state that the majority of the programs in the most common WCET benchmark suites do not meet EVT hypotheses, even in a simple and predictable architecture like the STM32 board. For what concerns the way execution times are measured, the two different methods proved to often lead to the same outcome, even if the employment of the STM32 internal timer looks preferable, since in some case the noise introduced by the logic analyser can worsen the outcome of the statistical tests. However, in some spare cases, the opposite can happen, with noise improving results by introducing variability when measurements from the timer are "stuck" into the same few values. Regarding the ε values, we discovered that, when fitting the GPD, for most of the benchmarks the shape parameter is negative (indicating a finite value of the tail), or at least close to 0 (which means the tail is infinite, but quickly converging to 0).

On the contrary, fitting the GEVD leads, in the majority of the cases, to positive values of ε. This is undesired when performing WCET analysis, since it indicates an unbounded tail, tending to 0 very slowly.

5.2 Modified benchmarks

Three of the benchmarks considered in the previous section (*fibcall*, *minver*, and *qsort-exam*) have been further modified and tested. The three benchmarks have been selected to cover different cases: in *fibcall*, the input parameter defines the number of iterations performed by the algorithm; in *minver*, it determines the size of the square matrix; in *qsort-exam*, it is the size of the input array to be sorted. In all the three modified programs, we considered five distribution classes to generate the input size: *exponential*, *normal*, *pareto*, *uniform*, and *weibull*.

5.2.1 Fibcall

The *fibcall* benchmark performs iterative Fibonacci calculations, and of course the higher is the position of the element of the series to be calculated, the longer the execution time of the benchmark is. In the original version, this parameter was determined in a random uniform manner, while in the new one, it is chosen by sampling from different distributions.

Results are shown in Table 5.3 and Table 5.4, where the modified versions are compared to the original one, respectively for the internal timer and the logical analyser cases. First, there is no difference between the outcomes of the logic analyser and internal timer, since the values they provide for PPI are very close. Furthermore, an improvement in the fulfilment of EVT hypotheses can be found in all modified benchmarks but *fibcall-uniform*, which gets the same result as *fibcall-original*. This makes sense, since also in the original version the input parameter determined the number of iterations, and the employed random function did it by sampling from uniform distribution as well. Thus, in this particular case, the difference is just in the function sampling the input, but the distribution shape is uniform in both cases.

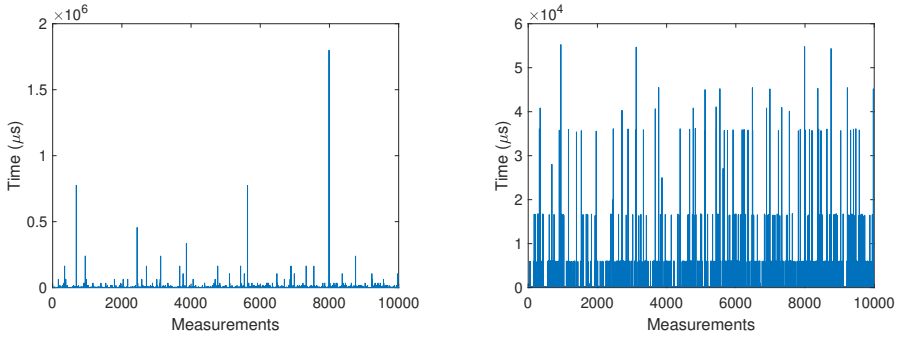
5.2.2 Minver

In the *minver* benchmark, the inverse of a floating-point matrix is computed. In the first version, only 3x3 matrixes were considered, and the only difference between one execution and the others is the initialization of the values in the matrixes themselves, performed each time completely at random. In the modified version, besides still initializing the matrix in a different way at every execution, we also vary the size according to the different kinds of distributions previously defined.

The related results are shown in Table 5.5 and Table 5.6. There are two cases in which the output derived from logic analyser's observations is different from the one obtained through internal timer's measurements. When determining the size of the matrix to be inverted through normal distribution, it is the internal timer that gets the higher value of the index, and the cause can still be found in the noise introduced by the logic analyser, which makes the hypotheses of *KPSS* and *h* not satisfied anymore. Vice versa, when dealing with the weibull distribution, the internal timer results lead to lower PPI value than the logic analyser. By looking at data related to these executions (whose plots are visible in Figure 5.2), it is clear that, in this case, the introduced noise is helpful for the statistical test to be passed. In fact, in the vast majority of the cases, measurement by the internal timer of the board result in the same 3 values of time, while there are just some spare cases varying from this behaviour. This is detected by the tests (in particular by *KPSS*, which carried out the lowest value) as evidence of a correlation between the different executions. On the contrary, this does not happen with the logic analyser, whose noise (which is not clearly visible in the plot, being most of the values very close to 0) introduces some kind of variability that makes the phenomenon invisible from the statistical tests standpoint.

5.2.3 Qsort-exam

The *Qsort-exam* benchmark implements the algorithm to perform the quick-sort of a floating-point array in a non-recursive way. Similarly to what happened in the *minver* benchmark, while in the original version it was just the content of the array to change between the various executions, in the modified one it is also the size that varies every time, always



(a) Logic analyser **(b)** Internal timer

Figure 5.2: *Plots of time series from "minver-weibull" benchmark, executed on the STM32 board, measuring time with a logic analyser (a) and the internal timer of the board (b).*

according to values taken from specific distributions.

Outcomes of the experiments are shown in Table 5.7 and Table 5.8. Results from the logic analyser do not show any difference when compared to the corresponding ones from the internal timer. Moreover, all modified versions obtain a high value of PPI, as it happened with the original one.

5.2.4 Final considerations

The set of modified benchmarks leads to very positive results in terms of time predictability. In general, the PPI value is on average at least as high as the one deriving from the original version of the code, and is improved in case the EVT hypotheses were not satisfied at the beginning. Furthermore, although there were some spare cases in which the outcome was lower than the critical value, all the benchmarks proved to be time predictable with at least one of the two ways to measure time.

	Board internal timer						
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
adpcm	0.938	0.891	0.957	0.944	0.914	0.874	-1.669
basicmath	0.948	0.891	0.992	0.914	0.938	0.207	0.169
bitcount	0.962	0.891	0.964	0.991	0.929	-0.339	-2.470
bsort100	0.920	0.891	0.949	0.906	0.905	-0.364	-2.633
cnt	0.914	0.891	0.899	0.930	0.915	-0.191	-2.992
compress	0.317	0.891	0.996	0.317	0.976	0.830	-2.085
coop	0.941	0.891	0.946	0.978	0.899	0.404	-0.505
cover	0.000	0.891	1.000	0.000	0.988	-1.207	-2.148
crc	0.931	0.891	0.953	0.910	0.930	-0.297	-2.323
duff	0.013	0.891	0.999	0.013	0.977	1.620	-1.679
edn	0.319	0.891	0.997	0.319	0.969	-0.856	-1.556
expint	0.952	0.891	0.961	0.999	0.896	-1.067	-1.244
fdct	0.958	0.891	0.981	0.947	0.945	-0.731	-2.486
fft1	0.945	0.891	0.974	0.950	0.909	-1.032	-1.702
fibcall	0.856	0.891	0.948	0.856	0.922	-0.457	-1.013
fir	0.015	0.891	0.998	0.015	0.970	-0.866	-1.516
fourierbench	0.237	0.891	0.966	0.237	0.910	-1.373	-2.329
insertsort	0.866	0.891	0.975	0.866	0.932	-0.245	-0.968
janne-complex	0.796	0.891	0.994	0.796	0.971	3.785	-0.267
jfdctint	0.405	0.891	0.999	0.405	0.984	-1.113	-2.324
lcdnum	0.380	0.891	1.000	0.380	0.987	0.266	-1.658
ludcmp	0.958	0.891	0.980	0.964	0.930	-0.151	-2.895
matmult	0.821	0.891	0.968	0.821	0.923	-0.028	-1.697
minver	0.954	0.891	0.981	0.958	0.922	-0.747	-2.644
ndes	0.078	0.891	0.996	0.078	0.980	-0.745	-1.927
prime	0.970	0.891	0.988	0.982	0.939	1.067	0.728
qsort-exam	0.955	0.891	0.991	0.942	0.933	-0.114	-1.321
qurt	0.947	0.891	0.990	0.924	0.926	-1.467	-1.830
recursion	0.955	0.891	0.969	0.976	0.921	2.691	2.219
select	0.963	0.891	0.989	0.957	0.942	-0.159	-0.646
shadriver	0.776	0.891	0.986	0.776	0.934	0.288	-2.024
sqrt	0.319	0.891	1.000	0.319	0.990	0.387	-2.082
statemate	0.053	0.891	1.000	0.053	0.993	-1.146	-1.661
ud	0.877	0.891	0.908	0.877	0.904	-0.022	-1.715

Table 5.1: *Results of original benchmarks, obtained by measuring time through the STM32 board internal timer.*

Logic Analyser Time							
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
adpcm	0.939	0.891	0.960	0.944	0.915	-1.426	-1.817
basicmath	0.949	0.891	0.984	0.935	0.930	-0.916	-1.098
bitcount	0.958	0.891	0.960	0.988	0.925	2.900	-0.025
bsort100	0.917	0.891	0.942	0.907	0.902	2.271	-0.494
cnt	0.234	0.891	0.257	0.928	0.803	3.306	-0.146
compress	0.315	0.891	0.939	0.315	0.911	1.619	-0.085
coop	0.941	0.891	0.946	0.978	0.899	0.495	0.054
cover	0.000	0.891	0.964	0.000	0.944	2.115	-0.011
crc	0.690	0.891	0.749	0.964	0.812	2.689	-0.058
duff	0.016	0.891	0.994	0.016	0.963	3.963	-0.138
edn	0.000	0.891	0.000	0.488	0.347	-1.190	-1.987
expint	0.952	0.891	0.961	0.999	0.896	0.537	0.164
fdct	0.705	0.891	0.775	0.915	0.801	2.544	-0.098
fft1	0.879	0.891	0.932	0.938	0.879	-0.316	-0.390
fibcall	0.857	0.891	0.949	0.857	0.922	0.000	-0.012
fir	0.000	0.891	0.001	0.005	0.507	-1.608	-1.707
fourierbench	0.252	0.891	0.966	0.252	0.907	-1.795	-1.969
insertsort	0.869	0.891	0.974	0.869	0.932	1.485	-0.025
janne-complex	0.839	0.891	0.996	0.839	0.964	4.003	2.710
jfdctint	0.391	0.891	0.917	0.391	0.913	3.064	-0.097
lcdnum	0.380	0.891	0.999	0.380	0.986	2.720	0.008
ludcmp	0.942	0.891	0.938	0.969	0.920	2.624	-0.305
matmult	0.000	0.891	0.000	0.000	0.212	-1.610	-1.398
minver	0.956	0.891	0.984	0.957	0.927	2.564	-0.105
ndes	0.000	0.891	0.002	0.206	0.554	3.367	-0.424
prime	0.970	0.891	0.988	0.983	0.939	0.035	0.000
qsort-exam	0.957	0.891	0.991	0.943	0.936	1.096	-0.066
qurt	0.947	0.891	0.990	0.924	0.927	1.170	0.921
recursion	0.927	0.891	0.919	0.965	0.898	1.127	1.046
select	0.962	0.891	0.989	0.956	0.942	-0.012	-0.063
shadriver	0.102	0.891	0.238	0.448	0.665	-1.580	-1.641
sqrt	0.317	0.891	0.998	0.317	0.987	2.256	0.004
statemate	0.053	0.891	0.998	0.053	0.975	1.129	-0.005
ud	0.880	0.891	0.921	0.880	0.910	2.322	-0.103

Table 5.2: *Results of original benchmarks, obtained by measuring time through a logic analyser.*

Board internal timer							
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
fibcall-original	0.856	0.891	0.948	0.856	0.922	-0.457	-1.013
fibcall-exponential	0.931	0.891	0.949	0.932	0.912	0.480	0.002
fibcall-normal	0.954	0.891	0.977	0.975	0.911	-0.220	-1.321
fibcall-pareto	0.934	0.891	0.936	0.950	0.916	0.601	0.195
fibcall-uniform	0.857	0.891	0.949	0.857	0.922	-0.456	-1.017
fibcall-weibull	0.958	0.891	0.978	0.977	0.920	1.910	1.252

Table 5.3: *Results of the modified versions of "fibcall" benchmark, obtained through the STM32 Board Internal Timer.*

Logic analyser Time							
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
fibcall-original	0.857	0.891	0.949	0.857	0.922	0.000	-0.012
fibcall-exponential	0.931	0.891	0.949	0.932	0.912	-0.018	-0.043
fibcall-normal	0.954	0.891	0.978	0.973	0.910	1.958	-0.046
fibcall-pareto	0.943	0.891	0.967	0.933	0.930	1.480	0.419
fibcall-uniform	0.857	0.891	0.948	0.857	0.922	0.000	-0.013
fibcall-weibull	0.954	0.891	0.958	0.989	0.915	0.297	0.177

Table 5.4: *Results of the modified versions of "fibcall" benchmark, obtained by measuring time through a logic analyser.*

Board internal timer							
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
minver-original	0.954	0.891	0.981	0.958	0.922	-0.747	-2.644
minver-exponential	0.941	0.891	0.963	0.929	0.932	1.968	1.476
minver-normal	0.958	0.891	0.983	0.968	0.924	-1.308	-2.014
minver-pareto	0.941	0.891	0.958	0.952	0.913	2.853	0.137
minver-uniform	0.964	0.891	0.987	0.975	0.930	-0.425	-0.650
minver-weibull	0.719	0.891	0.736	0.977	0.867	2.240	0.473

Table 5.5: *Results of the modified versions of "minver" benchmark, obtained through STM32 internal timer.*

Logic analyser Time							
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
minver-original	0.956	0.891	0.984	0.957	0.927	2.564	-0.105
minver-exponential	0.934	0.891	0.932	0.969	0.902	0.318	0.189
minver-normal	0.874	0.891	0.994	0.874	0.939	1.022	-0.226
minver-pareto	0.958	0.891	0.972	0.980	0.921	3.544	0.121
minver-uniform	0.944	0.891	0.979	0.920	0.934	1.298	0.992
minver-weibull	0.969	0.891	0.982	0.977	0.947	3.325	0.247

Table 5.6: *Results of the modified versions of "minver" benchmark, obtained by measuring time through a logic analyser.*

Board internal timer							
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
qsort-original	0.955	0.891	0.991	0.942	0.933	-0.114	-1.321
qsort-exponential	0.974	0.891	0.995	0.975	0.953	0.430	-0.166
qsort-normal	0.935	0.891	0.951	0.945	0.909	-0.212	-0.923
qsort-pareto	0.962	0.891	0.977	0.977	0.931	0.140	-0.141
qsort-uniform	0.946	0.891	0.927	0.990	0.920	-0.298	-0.697
qsort-weibull	0.953	0.891	0.969	0.961	0.929	0.314	-0.145

Table 5.7: *Results of the modified versions of "qsort" benchmark, obtained through STM32 internal timer.*

Logic analyser Time							
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
qsort-original	0.957	0.891	0.991	0.943	0.936	1.096	-0.066
qsort-exponential	0.974	0.891	0.995	0.975	0.953	0.783	-0.128
qsort-normal	0.933	0.891	0.947	0.944	0.907	-0.152	-0.357
qsort-pareto	0.961	0.891	0.977	0.977	0.931	1.201	-0.080
qsort-uniform	0.945	0.891	0.926	0.990	0.920	-0.074	-0.251
qsort-weibull	0.953	0.891	0.970	0.961	0.929	1.317	-0.103

Table 5.8: *Results of the modified versions of "qsort" benchmark, obtained by measuring time through a logic analyser.*

CHAPTER 6

The development of novel benchmarks

Due to the limitations of the previously described benchmarks, we developed new benchmarks from scratch, or derived them from the already existing ones by applying heavier changes (e.g. the code has been modified to become independent from the input values in terms of execution time, or alterations have been applied to whole parts of the original programs). The execution time is still measured through the internal timer of the board or the logic analyser, as described in the previous chapter.

6.1 Making the original benchmarks input-independent

Among the benchmarks taken into account in the previous chapter, there are twos sorting a numerical array: *qsort-exam*, already analysed by varying the size of the array in Section 5.2 and performing the quicksort algorithm, and *insertsort*, which orders the array by using the insertion sort technique, as the name suggests. The original versions of the two benchmarks have already been tested in Section 5.1. In this section, we, instead, explain how we developed new versions of them for further analysis. The

Algorithm 8: Input independent sorting benchmarks code

```
1  Function sort_indep():
2      random_seed := SEED;
3      array := [];
4      reversed := [];
5      for i in 1..length do
6          array[i] := random();
7          reversed[length - i + 1] := array[i];
8      end
9      start();
10     sort(array);
11     sort(reversed);
12     stop();
13 end
```

goal is to try to make the execution time less dependent on the input values, because it is obvious that a sorting algorithm takes much less time when dealing with an array that is almost sorted, rather than with a completely unordered one. This problem creates a dependency on the inputs, making the EVT hypotheses possibly invalid. To solve this problem, we implemented a simple strategy, i.e. sorting an array filled with the input values and its reversed version. In this way, even if the initial array is completely sorted, the best case (completely ordered array) is coupled with the worst one (completely unordered array) and vice versa. This makes the sorting strategy less dependent on the original inputs. Algorithm 8 shows the code used in this case.

Such a strategy obviously reduces the average performance by doubling the work to be performed. However, if it guarantees the pWCET to be computed, it can be beneficial for real-time systems. The seed is properly set to have the same input values as in the previous versions of the code. For what concerns the initialization part (whose instructions are still excluded from the time measurements), *line 5* fills the vector cell with a random number, as in the original benchmark, while *line 6* is added to fill the reversed array in specular order. At *lines 8-9*, the sorting algorithm is applied to both the previously built arrays. These instructions are included within the two functions (*start* and *stop*) used to measure the execution

Board internal timer							
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
insertsort-original	0.866	0.891	0.975	0.866	0.932	-0.245	-0.968
insertsort-indep	0.948	0.891	0.954	0.948	0.943	-0.318	-1.369
qsort-original	0.955	0.891	0.991	0.942	0.933	-0.114	-1.321
qsort-indep	0.922	0.891	0.918	0.947	0.901	-0.115	-1.941

Table 6.1: *Results from sorting benchmarks, obtained by measuring time through STM32 board internal timer.*

Logic analyser Time							
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
insertsort-original	0.869	0.891	0.974	0.869	0.932	1.485	-0.025
insertsort-indep	0.949	0.891	0.954	0.949	0.944	1.332	-0.030
qsort-original	0.957	0.891	0.991	0.943	0.936	1.096	-0.066
qsort-indep	0.915	0.891	0.902	0.944	0.898	1.223	-0.687

Table 6.2: *Results from sorting benchmarks, obtained by measuring time through a logic analyser.*

time.

The results are shown in Table 6.1 and Table 6.2. This kind of approach is very effective for *insertsort* benchmark, whose input-independent version fulfils EVT hypotheses, unlike the original version of the code. For what concerns *qsort*, instead, there is not much difference between the two versions: in both cases PPI value is higher than the critical threshold, even if the independent version of the benchmark leads to slightly lower output. The modifications to make the two benchmarks independent from the input values have a cost in terms of execution times, which is 2 times higher in *qsort* and 2.5 in *insertsort*. However, it guarantees the satisfaction of the EVT hypotheses, independently on the input data. This achievement is more clear when the input data is not random, as shown in the next section.

6.2 TCAS-sort

Instead of completely random generated values, from this section we present the experiments using real applications and data, to simulate a realistic result. The goal is to assess the differences in the results obtained

from the two different approaches. For this purpose, the preliminary stages of the *TCAS* algorithm, which is described in detail in Section 6.5, have been chosen as the candidate for the first experiment. TCAS is the acronym for *Traffic Alert and Collision Avoidance System* and is a software used on aircraft, that provides advice to the pilots about the manoeuvres to be performed to do not enter in dangerous situations, potentially leading to a collision. As we later describe in detail, the TCAS software analyses the position of nearby aircraft. There may be more than one *intruder* w.r.t. the position of the aircraft running the algorithm, and the closer it is, the higher is the priority with which its status has to be checked. To prioritize the aircraft, a sorting algorithm is employed: the inputs to the program are the coordinates (latitude and longitude, altitude is not considered at this stage) of the aircraft running the software and all the intruders. At each execution, at most 20 intruders can be taken into account, but in most of the cases, the limit number of aircraft is not reached, because not so much aircraft is "visible" by the TCAS antennas. In the first positions of the array, the computed distances are inserted, while the empty positions are filled with fictitious aircraft at "infinite" distance, so that the input appears with the same size. To sort the array, the *bubble-sort* algorithm has been chosen. The code of the benchmark can be found in Algorithm 9.

In the initialization part (*lines 1-8*), the distances between the aircraft (whose coordinates are stored in *my_lat* and *my_lon* variables) and the intruders (whose positions can be found into arrays *lat* and *lon*) are computed. As already explained, when data is missing (value indicated as *X* in the code), distance is set to infinite. This part of the computation is excluded from the measurements of execution time, since it is not the object of the analysis, and it should not affect the results obtained through the sorting algorithm. The input data have been extracted from a flight simulator.

The overall structure of the code is similar to the previous algorithms. What changes between the input-independent version of the code and the original one is that the input array is sorted together with its reverse, as it happened in the case of Section 6.1, and, most importantly, the way *bubblesort* algorithm is implemented. This second difference is shown in Algorithm 10 and Algorithm 11.

In the original version, the sorting algorithm is fully optimized: the

Algorithm 9: TCAS-sort benchmark code

```
1  Function TCAS_sort (my_lat, my_lon, lat[], lon[]) :
2      distances := [];
3      for i in 1..20 do
4          if lat[i] == X then
5              distances[i] := ∞;
6          end
7          else
8              distances[i] := √((my_lat − lat[i])² + (my_lon − lon[i])²);
9          end
10     end
11     start();
12     bubblesort(distances);
13     stop();
14 end
```

array is fully scanned only once, then, at each iteration, it is possible to check one position less than the previous iteration, since we know that every time the highest element (among the unordered ones) will be moved to the right position. Furthermore, at each cycle, we check that at least one swap is performed: when this does not happen, it means the array is sorted and the process can stop. This makes the execution time input-dependent.

In the input-independent version, the first optimization, exploiting the fact that it is possible to check one position less of the array than the previous iteration, is still present. On the contrary, the second one, checking if the process can be stopped due to already sorted array, has been removed, causing the same number of cycles to be performed (since the length of the array does not vary) independently from the input.

The outcome, shown in Table 6.3 and Table 6.4, proves the modifications are very effective in this case. With both timing measurement approaches (i.e. with the logic analyser and internal timer), the value of PPI is much higher with the independent version than with the original one. In the independent case, EVT hypotheses are clearly respected, while in the dependent one the outcome is very close to the critical value, but not high enough to state their fulfilment. The differences between the two versions of the code are also visible in Figure 6.1, showing the plots of

Algorithm 10: Bubblesort implementation - original version

```
1  Function bubblesort(array):
2      swapped := 1;
3      end := array.length - 1;
4      while swapped = 1 do
5          swapped := 0;
6          for i in 1..end do
7              if array[i] > array[i+1] then
8                  swap(array[i], array[i+1]);
9                  swapped := 1;
10             end
11         end
12         end–;
13     end
14 end
```

the time series obtained through the internal timer. In the original code, a dependence on the number of intruders is evident. This is expected, since the remaining positions of the array are filled with "infinite" values, which are already in the right position and do not need to be swapped. This improvement in timing predictability does not come for free: the independent version leads to execution times that are on average 2.9 times higher than the ones from the original program.

6.3 A loop-based benchmark

In order to test the satisfaction of EVT hypotheses when the execution time follows a given distribution, we created a trivial benchmark that consists of a loop, and it is composed of very basic operation, such as the increment of a variable. The number of times the cycle is visited is not fixed, but randomly determined at each execution. Similarly to what is already done in Section 5.2, the number of iterations is determined by sampling values from different kinds of distributions: *uniform*, *exponential*, *pareto*, *weibull*, and *normal*. The structure of the code can be found in Algorithm 12. *Line 2* is where the number of iterations is determined: it is sufficient to change the function called for getting the random value (one for each different

Algorithm 11: Bubblesort implementation - independent version

```
1  Function bubblesortIndep(array):
2      end := array.length - 1;
3      while end ≥ 1 do
4          for i in 1..end do
5              if array[i] > array[i+1] then
6                  swap(array[i], array[i+1]);
7              end
8          end
9          end–;
10     end
11 end
```

	Board internal timer						
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
tcas-sort	0.000	0.891	0.000	0.000	0.137	-0.589	-1.022
tcas-sort-indep	0.963	0.891	0.973	0.973	0.942	0.274	-2.030

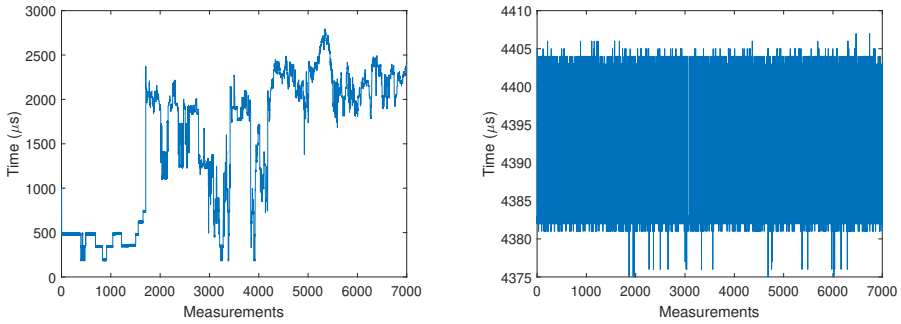
Table 6.3: *Results from "tcas-sort" benchmark, obtained by measuring time through STM32 board internal timer.*

distribution has been created) to have a "new" benchmark (i.e. benchmark doing the same thing, but with different behaviour in terms of number of cycles). This operation is not included in the time measurements, since they begin at *line 3* and end at *line 7*, after the for loop has been completed.

Tables 6.5 and 6.6 show the outcomes of the described programs, with the two different ways of making time measurements. The names simply refer to what kind of distribution has been used to determine the number of times the *for* loop is iterated. All the distributions show good results, since in all the cases the internal timer and the logic analyser provide very close values for all the different tests performed on data. PPI values are generally very high, and greater than the critical one in all cases but when the uniform distribution is employed.

	Logic analyser Time						
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
tcas-sort	0.000	0.891	0.000	0.000	0.137	2.496	2.794
tcas-sort-indep	0.861	0.891	0.875	0.970	0.875	4.320	1.523

Table 6.4: *Results from "tcas-sort" benchmark, obtained by measuring time through a logic analyser.*



(a) Original version

(b) Input-independent version

Figure 6.1: *Plots of time series from the two versions of "tcas-sort" benchmark, original (a) and input-independent (b), employing the internal timer for the measurements.*

6.4 Audio compression benchmark

Starting from the *timing* benchmark included into *Mediabench* suite, we developed a novel audio compression benchmark. The behaviour of the original benchmark was to read data from a *.wav* file, compress it to have half of the space occupied, and then do the opposite operation, i.e. to re-build the original audio file after decompression. In the modified version of the code, all those parts involving *.wav* files (both in the data retrieving and audio file re-generation phases) have been removed, focusing on the pure compression and decompression tasks. The input and output data format has been changed from *.wav* files to arrays of *short integers*, so that retrieving data directly from the file (which cannot be done by the STM32 board) is not necessary. Even in this case, the resulting benchmark has both been studied with random and real inputs. In the first case, the previously described random generation function has been used, while, in

Algorithm 12: Trivial benchmarks code

```
1 x := 0;
2 cycles := sample_from_distribution();
3 start();
4 for i in 1 .. cycles do
5 |   x++;
6 end
7 stop();
```

Board internal timer

BENCHMARK	ppi	cval	kpss	bds	h	ε	ε pareto
exponential	0.926	0.891	0.967	0.891	0.921	0.356	-0.232
normal	0.938	0.891	0.969	0.947	0.898	-0.230	-0.876
pareto	0.953	0.891	0.973	0.980	0.907	0.643	0.295
uniform	0.857	0.891	0.948	0.857	0.922	-0.457	-1.021
weibull	0.936	0.891	0.976	0.932	0.902	1.673	0.963

Table 6.5: *Results from trivial benchmarks, obtained by measuring time through the STM32 board internal timer.*

the second case, the original version of the program was employed: the instructions reading the input file and turning it into a numerical arrays were used to scan chunks of 100 bytes of a real audio file, and exporting them into a text file that could then be used as an input source for the modified version of the benchmark.

Table 6.7 and Table 6.8 contain the results obtained by running *timing* benchmark. The suffixes in the name explain which kind of data has been used as input (*randomval* indicates data generated through a random function, while in *realval* it has been derived from an existing audio file). The comparison between the two different tables shows the two different ways of measuring time lead to very similar results in terms of PPI. It is interesting to notice there is a huge difference between the values of the index when using random input, with respect to when a real use of the application is simulated. In the first case, all the hypotheses are respected, and so the test is passed, while in the second case the values of PPI are very low, and none of the outcomes of the three tests checking independence in the data is above the critical threshold. This difference is

Logic analyser Time							
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
exponential	0.932	0.891	0.950	0.933	0.912	0.462	0.151
normal	0.938	0.891	0.969	0.947	0.898	-0.016	-0.105
pareto	0.943	0.891	0.967	0.933	0.930	0.814	0.318
uniform	0.856	0.891	0.948	0.856	0.922	0.667	-0.036
weibull	0.954	0.891	0.958	0.989	0.915	0.740	0.659

Table 6.6: *Results from trivial benchmarks, obtained by measuring time through a logic analyser.*

Board internal timer							
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
timing-randomval	0.954	0.891	0.960	0.952	0.949	-0.238	-2.741
timing-realval	0.093	0.891	0.253	0.353	0.684	-0.307	-2.802

Table 6.7: *Results from audio compression benchmark, obtained by measuring time through the STM32 board internal timer.*

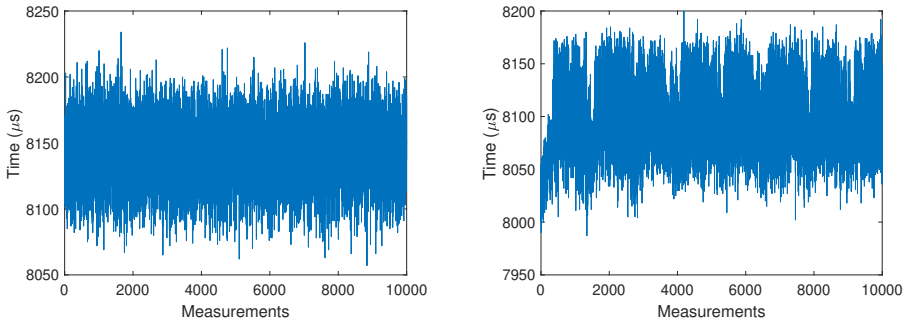
clearly visible also from the plots in Figure 6.2, that show the data taken from the internal timer of the board: the increasing trend, that is evident in the first executions with real values, is not present with randomly generated values, and is one of the causes determining such a high difference in the results. Thus, this can be taken as an example of the importance of the concept of representativity, as explained in Section 2.1, which is critical when performing probabilistic timing analysis: in this case, studying the benchmark by just submitting randomly generated numerical arrays would have brought to draw conclusions not reflecting the actual behaviour of the program in a real-world use case (i.e. when compressing and then bringing back to the original format a real audio file).

6.5 TCAS

The *Traffic Alert and Collision Avoidance System* (TCAS) is a software helping aircraft pilots to avoid dangerous manoeuvres that could potentially lead to a collision with other aircraft flying nearby. The preliminary stage of the algorithm, ordering up to 20 intruders according to their distance from the considered plane to assign higher priority to the closest

Logic analyser Time							
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
timing-randomval	0.937	0.891	0.972	0.935	0.905	-1.537	-1.668
timing-realval	0.157	0.891	0.448	0.367	0.658	2.921	0.106

Table 6.8: *Results from audio compression benchmark, obtained by measuring time through a logic analyser.*



(a) Random values **(b)** Real values

Figure 6.2: *Plots of time series from "timing" benchmark, employing the internal timer of the board for the measurements and using random values (a) and real values (b) as inputs.*

ones, has already been discussed in Section 6.1. The second part of the program consists of deciding what kind of action the pilots must be advised to perform, in order to keep the safety conditions (i.e. a minimum distance from the other aircraft) respected. There are some implementations available on the web for this task that have been also used as WCET benchmark, however, the found ones resulted very simple and not realistic. This simplicity causes them to have too few conditional branches determining different kinds of behaviour when different input values are submitted. We decided then to write a new version, in order to create a benchmark closer to a real application, following the description of the software in [25]. There are 6 possible outputs the algorithm can return: *climb* and *climb at high speed*, meaning the altitude of the aircraft must be incremented (the speed is determined according to how high is the danger of collision with the intruder); *descend* and *descend at high speed*, similar to the previous two, but shown when the altitude must be decreased;

Board internal timer							
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
tcas-randomval	0.954	0.891	0.985	0.933	0.942	1.349	-0.547
tcas-realval	0.000	0.891	0.047	0.000	0.512	0.042	-0.514

Table 6.9: *Results from TCAS benchmark, obtained by measuring time through the STM32 board internal timer.*

Logic analyser Time							
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
tcas-randomval	0.954	0.891	0.985	0.933	0.942	0.420	-0.120
tcas-realval	0.000	0.891	0.000	0.978	0.381	1.222	0.716

Table 6.10: *Results from TCAS benchmark, obtained by measuring time through a logic analyser.*

traffic, indicating there is other aircraft nearby, but no manoeuvre is needed to avoid it; *clear*, stating there is nothing in the surroundings. For the actual software to work properly, each aircraft should send information related to its flight status to the others, and receive the corresponding data from the others. However, this part has not been implemented in our benchmark, supposing the necessary information has already been retrieved. Furthermore, the created code compares the status of the aircraft with the one of the intruders, an operation that needs to be repeated for each airplane flying nearby (following the priorities established in the preliminary phase) in the real application. The input variables to the algorithm are the coordinates of the two aircraft (in terms of latitude, longitude and altitude), their horizontal speeds (split into x and y axes), and the vertical ones. The algorithm has both been tested through randomly generated input and realistic one, obtained with a flight simulator software.

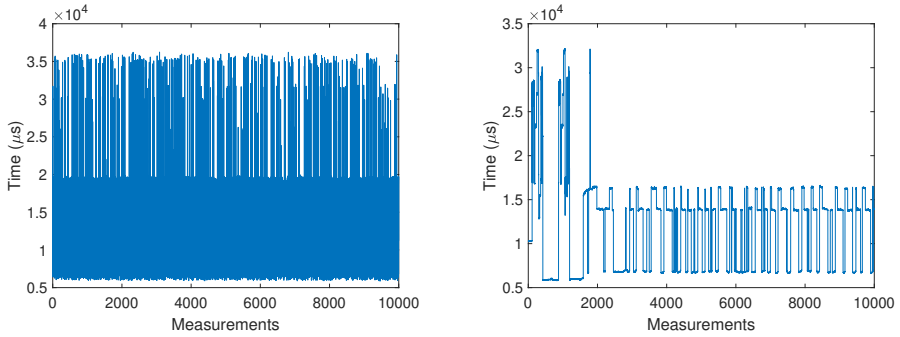
In Table 6.9 and Table 6.10, the outcomes of the performed experiments are shown. As it happened in the previous case, the values of the probabilistic predictability index are very close between the internal timer and logic analyser. Instead, there is a big difference between the outputs coming from random input, that proves to respect the hypotheses of PPI tests, and the ones derived from real data, that lead to a very low PPI value (with the only difference that *bds* test passed with the logic analyser, while

it does not with the internal timer). The reason of the different results is clearly visible in Figure 6.3: with random values, the program keeps behaving the same way from the beginning to the end, while real data show different functioning modes through the 10.000 measurements, this leading to the very low value for the PPI test, as previously seen. For further analysis, a *change detection algorithm* has been written and employed to split the set of measurements into chunks showing the same behaviour. The algorithm works as follows: first, it loads a chunk of 20 execution time values, computing its mean and standard deviation. After that, the following measurements are taken one per one and added to the to the current chunk (every time updating the previously mentioned statistical values) unless one of the following two conditions (inspired to the Western Electric rules) occurs:

- the difference between the following value and the current mean is higher than three times the current standard deviation;
- at least two out of the three following values differ from the current mean by more than two times the current standard deviation.

When it happens, it means a change of behaviour has been detected. Thus, the previous chunk is completed, and the process restarts by creating a new one composed by the following 20 measurements. The first 20 elements considered when a new chunk is started, however, may contain 2 (or more) different modes, which is undesired for our analysis. To prevent this from happening, the ratio of the first 20 elements of the chunk and their standard deviation is computed and, in case it is lower than 2 (indicating too high standard deviation value with respect to the average), the window including the values for the initialization is shifted by one, in order to exclude the first element and substitute it with the 21st. The process is iterated until the condition on the ratio is matched.

Running the algorithm on previously shown data, 223 different modes are detected. PPI test can then be performed on each of the chunks the set of data has been split into. The results are shown in Figure 6.4: the average value of PPI is 0.759, which is much higher than the one observed before, while the median is 0.814. Furthermore, in 56 out of the 223 cases (i.e. about the 25% of the found chunks) the value is higher than the critical one, meaning the hypotheses are fulfilled. In any case, the



(a) Random values **(b)** Real values

Figure 6.3: *Plots of time series from "TCAS" benchmark, employing the internal timer for the measurements, and using random values (a) and real values (b) as inputs.*

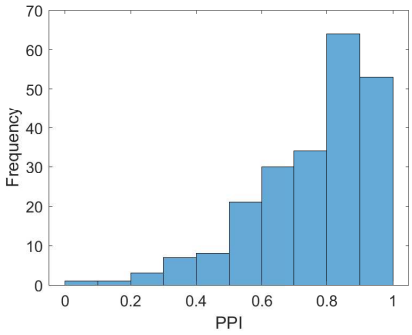


Figure 6.4: *Plot of the PPI values of the different chunks of data of "TCAS" benchmark.*

outcome we get with realistic input is still different than what we would have concluded by just observing results provided by random data. Thus, after the *timing* benchmark described in Section 6.4, this is another proof that when dealing with a real application, it is very important to study it using inputs reflecting as much as possible its actual operating behaviour. The detection and identification of the modes enable to split the input values to different datasets, and they help to satisfy the PPI hypotheses and, consequently, to produce a correct pWCET.

CHAPTER 7

Experimental evaluation of the Linux real-time patch

In this Chapter, results obtained through the employment of a Raspberry PI 4 are presented. As already explained in Section 4.4.2, two different configurations have been tested: benchmarks have been run on the standard command-line Linux version *Raspberry Pi OS 1.4*, and then they have been re-executed after the application of *PREEMPT_RT Patch*. The latter increases time predictability by making the kernel code preemptable (with the exception of some small critical regions), while lowering the average throughput of the system.

7.1 PREEMPT_RT: The Linux real-time patch

The main goal of the *PREEMPT_RT* patch is to increase the preemptibility degree of the kernel code, in order to increase its predictability and reduce the latencies. It is described in detail by Reghenzani et al. in [28], and its main features are:

- *High-resolution timers* (HRTs): the patch allows to instantiate timers with a resolution in the order of nanoseconds (the actual resolution depends on the hardware implementation). They can be enabled or disabled at compile-time via the *CONFIG_HIGH_RES_TIMERS* configuration flag, or at runtime. HRTs also enable *Full Tickless Operation*: in the Linux kernel, in fact, an interrupt is triggered at a fixed rate to update the internal structures. This recurrent event can be disabled with the introduction of HRTs, thus decreasing extra latencies due to the preemptions caused by the periodic timer;

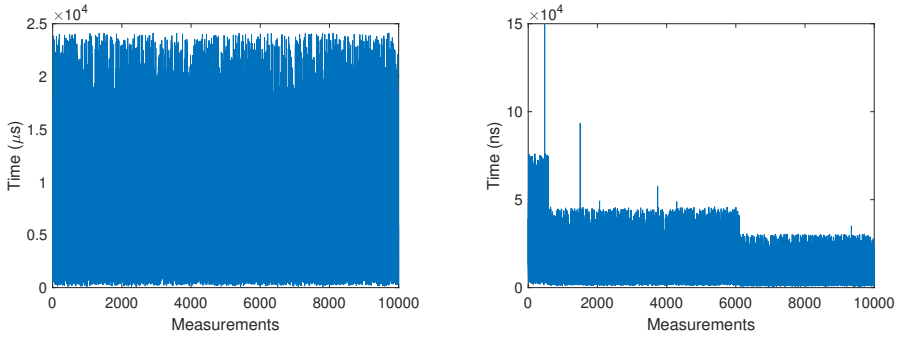
- *Threaded Interrupt Handlers*: in Linux, each interrupt handler is split into *top-half* part (*hard IRQ*), quickly reacting to the interrupt and performing the most critical operations, and *bottom-half* part (*soft IRQ*), performing additional computation. With *PREEMPT_RT* patch, the top-half part can be written as a simple function waking up the related interrupt thread, thus scheduling the bottom half part as a regular kernel thread;

- *Priority Inheritance*: it is a solution to the well-known *priority inversion* problem, in which a thread with lower priority may indirectly preempt a higher-priority one. The idea is to increase the priority of the low-level tasks locking resources required by higher-level ones to the same level of the highest-priority task among the ones requiring the same resource.

- *Read-Copy-Update (RCU)*: this synchronization technique is particularly efficient in scenarios involving multiple readers and a single writer. Standard read-write locks are replaced by RCU primitives, thus preventing read-side locks from blocking, and creating multiple versions of the resource updated by the writers;

- *Memory Allocators*: PREEMPT_RT employs *Two-Level Segregate Fit* (TLSF) approach for memory allocation. Its main feature is the capability to allocate memory in $O(1)$ complexity. This allows having bounded WCET for memory allocation operations, independently from applications and data.

- *Preemptible Spinlocks*: they are a locking mechanism where a thread continuously attempts to acquire the lock until performing it suc-



(a) STM32 board internal timer

(b) Raspberry PI4 - Standard Linux

Figure 7.1: *Comparison of time series from "coop" benchmark, executed on the STM32 board (a), and on the Raspberry PI4 (b).*

cessfully. This "spinning" causes a waste of CPU time and prevents the possibility of preemption, however, it results in a benefit for the overall performance. With the *PREEMPT_RT* patch, most of the non-preemptible spinlocks are removed.

7.2 Experimental Results

7.2.1 Plain Linux

The configuration employing a Raspberry PI 4 running the standard command line Linux OS (whose results are visible in Tables 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, and 7.9) proves not to be a good setup for time predictability. In fact, among the 60 benchmarks that have been tested, just 8 (i.e. about 13%) proved to respect EVT hypotheses. The reason why this happens can be found in the high amount of dependencies and unpredictability of the Linux kernel. This is clearly visible in Figure 7.1, comparing the plots obtained from the internal timer of the STM32 board and from Raspberry PI 4 with standard Linux OS. In the former, observations keep the same behaviour through all the 10.000 executions, while in the latter at least three different modes are present, together with some measurements higher than the general trend, caused by the previously described unpredictable latencies.

7.2.2 PREEMPT_RT Linux

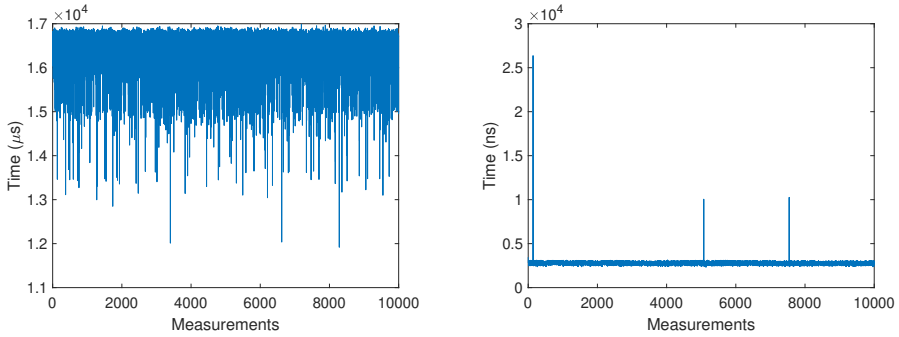
Results deriving from the application of *PREEMPT_RT patch* (shown in Tables 7.10, 7.11, 7.12, 7.13, 7.14, 7.15, 7.16, 7.17, and 7.18) look comparable with the ones we obtained through the STM32 board. However, there are some benchmarks in which the outcome the two architectures lead to is different. In fact, when using the Raspberry PI, the execution times still suffer from unpredictability: from the software side, this is due to the fact that the board relies on an OS, which introduces some spurious latencies (e.g. preemption and interrupts); instead, for what concerns the hardware, high-latency events are mainly given by the CPU complexity, for instance, the fact that it is composed of four cores. Thus, different tasks can be executed at the same time, which may cause interferences between themselves.

Unexpected latencies have two different kinds of effects. Sometimes they cause EVT hypotheses, that were matched when using STM32 board, not to be respected anymore. This happens for example in benchmark *minver*, whose plots can be found in Figure 7.2. Latency peaks are clearly visible in those execution times being much higher than the average, and outputs of statistical tests show they result in a significant decrease of PPI value. On the contrary, in case a program showed too low variability when executed on the STM32 board, introducing some noise can prevent statistical tests from detecting dependence among executions, and thus from failing, in a similar way as we have already observed when comparing the internal timer of the board with the logic analyser. This happens for instance in benchmark *edn*, which can be found in Figure 7.3.

7.2.3 Comparison & Discussion

Besides the characteristics pointed out in Section 7.1, another difference between the two configurations the Raspberry PI 4 has been employed with is that, with PREEMPT_RT Patch, all interrupts have been moved to a different core than the one on which the benchmarks were executed (excluding the few for which this was not possible). On the other hand, with standard Linux, interrupts were left on the core they had been assigned to by default.

All the aforementioned dissimilarities result in a decrease in the general

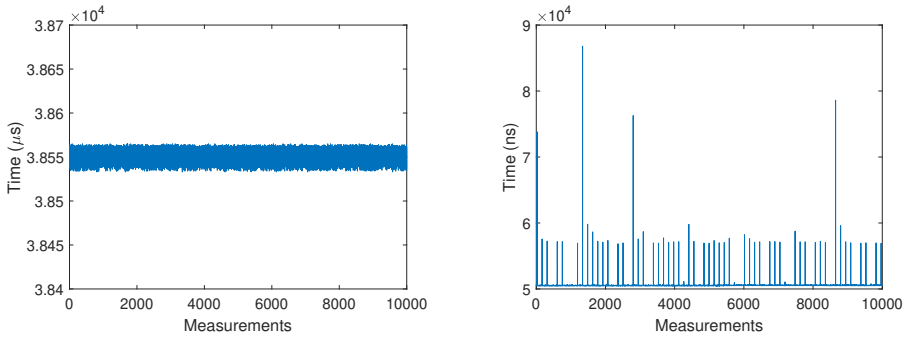


(a) STM32 board internal timer

(b) Raspberry PI4 - PREEMPT_RT patch

Figure 7.2: *Comparison of time series from "minver" benchmark, executed on the STM32 board (a), and on the Raspberry PI4 with PREEMPT_RT patch application (b).*

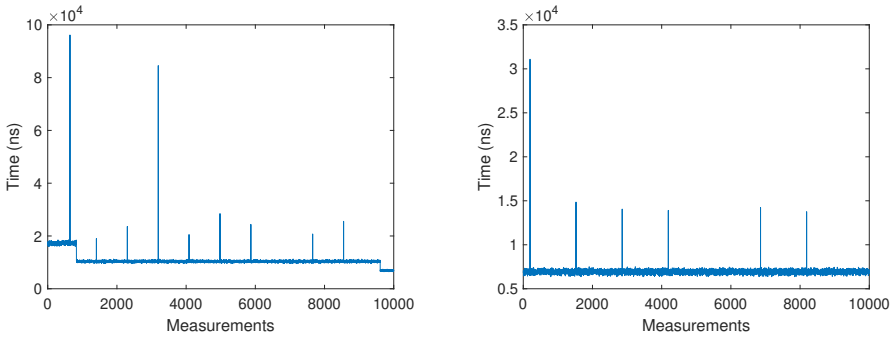
throughput of the system when the PREEMPT_RT patch is applied, due to the higher number of context switches. However, they introduce an improvement in terms of time predictability. This is because the frequency at which the execution of the programs is stopped (due to preempting tasks or interrupts) is significantly lowered. This phenomenon is shown in Figure 7.4, where plots related to *bitcount* benchmark are visible: outliers happen more often when the program is run on the standard Linux version (whose graph also presents three different modes, unlike the one deriving from Real-Time patch). Seldom, however, we observe the opposite, i.e. the highest value of PPI is obtained when the PREEMPT_RT patch is not applied. This is, for instance, the case of *qsort-exam* benchmark, shown in Figure 7.5. Outliers are still more frequent in the plot from the standard Linux version, yet it presents the same behaviour throughout the experiments, unlike in the previously mentioned program. Statistical tests state in this case EVT hypotheses are respected only when PREEMPT_RT Patch is not applied.



(a) STM32 board internal timer

(b) Raspberry PI4 - PREEMPT_RT patch

Figure 7.3: *Comparison of time series from "edn" benchmark, executed on the STM32 board (a), and on the Raspberry PI4 with PREEMPT_RT patch application (b).*



(a) Standard Linux

(b) PREEMPT_RT patch

Figure 7.4: *Comparison of time series from "bitcount" benchmark, executed on a Raspberry PI4 with standard Linux version (a), and PREEMPT_RT patch application (b).*

Linux - original							
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
adpcm	0.000	0.891	0.000	0.003	0.483	0.200	-0.483
basicmath	0.000	0.891	0.000	0.003	0.502	0.247	-0.246
bitcount	0.000	0.891	0.000	0.001	0.222	0.009	-0.119
bsort100	0.000	0.891	0.000	0.001	0.183	0.384	-0.415
cnt	0.000	0.891	0.000	0.000	0.182	0.642	-0.209
coop	0.000	0.891	0.000	0.524	0.437	0.432	-0.107
cover	0.000	0.891	0.000	0.001	0.188	0.833	-0.231
crc	0.000	0.891	0.000	0.997	0.321	0.791	-0.049
duff	0.012	0.891	0.016	0.999	0.624	1.623	0.000
edn	0.000	0.891	0.000	0.001	0.137	0.873	-0.364
expint	0.000	0.891	0.000	0.974	0.428	0.027	-0.025
fdct	0.000	0.891	0.000	0.001	0.323	-0.030	-0.153
fft1	0.000	0.891	0.000	0.001	0.267	2.234	-0.049
fibcall	0.008	0.891	0.016	0.598	0.611	-0.020	-0.072
fir	0.000	0.891	0.000	0.002	0.399	0.288	-0.412
fourierbench	0.000	0.891	0.000	0.001	0.189	0.452	-0.419
insertsort	0.000	0.891	0.000	0.041	0.126	-0.001	-0.128
janne-complex	0.004	0.891	0.006	0.999	0.604	0.023	-0.007
jfdctint	0.965	0.891	0.974	0.999	0.921	3.532	-0.041
lcdnum	0.792	0.891	0.846	0.829	0.893	0.022	-0.066
ludcmp	0.000	0.891	0.000	0.000	0.098	0.625	-0.187
matmult	0.000	0.891	0.000	0.001	0.215	0.885	-0.434
minver	0.000	0.891	0.000	0.008	0.236	0.039	-0.095
ndes	0.000	0.891	0.000	0.000	0.097	0.869	-0.253
prime	0.001	0.891	0.001	0.991	0.526	0.732	0.036
qsort-exam	0.960	0.891	0.942	0.996	0.940	0.033	-0.036
qurt	0.958	0.891	0.937	0.999	0.937	0.018	-0.022
recursion	0.305	0.891	0.352	0.922	0.759	4.075	1.433
select	0.952	0.891	0.971	0.947	0.937	0.009	-0.024
shadriver	0.000	0.891	0.000	0.002	0.283	0.636	-0.463
sqrt	0.000	0.891	0.000	0.063	0.175	0.032	-0.034
statemate	0.000	0.891	0.000	0.518	0.099	0.042	-0.036

Table 7.1: *Results from original benchmarks, run on a Raspberry PI 4 with standard Linux version.*

Linux - fibcall

BENCHMARK	ppi	cval	kpss	bds	h	ε	ε pareto
fibcall-original	0.008	0.891	0.016	0.598	0.611	-0.020	-0.072
fibcall-exponential	0.000	0.891	0.000	0.000	0.089	-0.399	-1.345
fibcall-normal	0.621	0.891	0.863	0.737	0.757	-0.033	-1.556
fibcall-pareto	0.000	0.891	0.000	0.422	0.196	0.029	-0.023
fibcall-uniform	0.971	0.891	0.977	0.999	0.936	3.833	-0.092
fibcall-weibull	0.000	0.891	0.000	0.000	0.100	5.346	-0.071

Table 7.2: *Results from modified versions of "fibcall" benchmark, run on a Raspberry PI 4 with standard Linux version.*

Linux - minver

BENCHMARK	ppi	cval	kpss	bds	h	ε	ε pareto
minver-original	0.000	0.891	0.000	0.008	0.236	0.039	-0.095
minver-exponential	0.000	0.891	0.000	0.000	0.067	1.716	-0.053
minver-normal	0.380	0.891	0.414	0.999	0.809	0.016	-0.004
minver-pareto	0.971	0.891	0.978	0.999	0.936	0.069	-0.005
minver-uniform	0.017	0.891	0.039	0.482	0.615	0.013	-1.483
minver-weibull	0.000	0.891	0.000	0.000	0.054	-0.383	-1.199

Table 7.3: *Results from modified versions of "minver" benchmark, run on a Raspberry PI 4 with standard Linux version.*

Linux - qsort

BENCHMARK	ppi	cval	kpss	bds	h	ε	ε pareto
qsort-original	0.960	0.891	0.942	0.996	0.940	0.033	-0.036
qsort-exponential	0.000	0.891	0.000	0.000	0.055	0.727	-0.266
qsort-normal	0.000	0.891	0.000	0.000	0.075	0.277	-0.082
qsort-pareto	0.001	0.891	0.015	0.009	0.635	0.189	-0.047
qsort-uniform	0.000	0.891	0.000	0.073	0.204	0.130	-0.075
qsort-weibull	0.000	0.891	0.000	0.709	0.169	0.035	-0.026

Table 7.4: *Results from modified versions of "qsort" benchmark, run on a Raspberry PI 4 with standard Linux version.*

Linux - sorting

BENCHMARK	ppi	cval	kpss	bds	h	ε	ε pareto
insertsort-original	0.000	0.891	0.000	0.041	0.126	-0.001	-0.128
insertsort-indep	0.000	0.891	0.000	0.001	0.131	0.252	-0.078
qsort-original	0.960	0.891	0.942	0.996	0.940	0.033	-0.036
qsort-indep	0.000	0.891	0.000	0.001	0.084	0.735	-0.321

Table 7.5: *Results from sorting benchmarks, run on a Raspberry PI 4 with standard Linux version.*

Linux - TCAS-sort

BENCHMARK	ppi	cval	kpss	bds	h	ε	ε pareto
tcas-sort	0.000	0.891	0.000	0.000	0.401	0.064	-0.001
tcas-sort-indep	0.000	0.891	0.000	0.000	0.106	1.074	-0.053

Table 7.6: *Results from "TCAS-sort" benchmark, run on a Raspberry PI 4 with standard Linux version.*

Linux - trivial

BENCHMARK	ppi	cval	kpss	bds	h	ε	ε pareto
exponential	0.000	0.891	0.000	0.607	0.472	0.546	0.077
normal	0.000	0.891	0.000	0.025	0.107	0.016	-0.363
pareto	0.913	0.891	0.907	0.930	0.902	1.099	0.442
uniform	0.000	0.891	0.000	0.217	0.294	-0.006	-0.183
weibull	0.931	0.891	0.896	0.989	0.909	2.049	1.376

Table 7.7: *Results from trivial benchmarks, run on a Raspberry PI 4 with standard Linux version.*

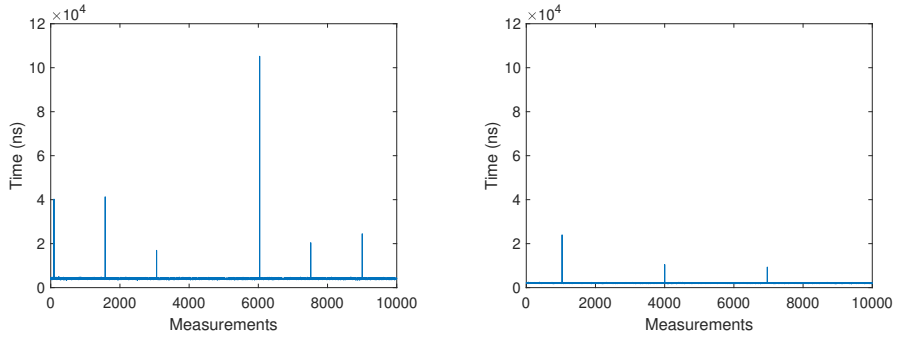
Linux - TCAS

BENCHMARK	ppi	cval	kpss	bds	h	ε	ε pareto
tcas-randomval	0.870	0.891	0.988	0.870	0.927	0.272	-0.025
tcas-realval	0.000	0.891	0.000	0.001	0.301	0.402	-0.013

Table 7.8: *Results from "TCAS" benchmark, run on a Raspberry PI 4 with standard Linux version.*

Linux - timing							
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	ε	ε **pareto**
timing-randomval	0.000	0.891	0.000	0.000	0.054	1.015	-0.171
timing-realval	0.000	0.891	0.000	0.000	0.169	0.000	-0.082

Table 7.9: *Results from audio compression benchmark, run on a Raspberry PI 4 with standard Linux version.*



(a) Standard Linux **(b)** PREEMPT_RT patch

Figure 7.5: *Comparison of time series from "qsort-exam" benchmark, executed on a Raspberry PI4 with standard Linux version (a), and PREEMPT_RT patch application (b).*

PREEMPT_RT - original							
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	**ε**	**ε pareto**
adpcm	0.848	0.891	0.943	0.848	0.897	0.068	-0.460
basicmath	0.981	0.891	0.987	0.996	0.960	0.126	0.019
bitcount	0.961	0.891	0.952	0.996	0.935	0.010	-0.252
bsort100	0.000	0.891	0.331	0.000	0.612	-0.190	-1.354
cnt	0.455	0.891	0.986	0.455	0.946	-0.100	-2.664
coop	0.942	0.891	0.949	0.977	0.899	0.286	-0.203
cover	0.971	0.891	0.972	0.997	0.945	0.076	-0.216
crc	0.975	0.891	0.985	0.998	0.941	0.084	-0.142
duff	0.492	0.891	0.522	0.999	0.834	0.049	-0.058
edn	0.973	0.891	0.992	0.970	0.958	0.140	-0.875
expint	0.957	0.891	0.986	0.951	0.934	0.400	-0.700
fdct	0.872	0.891	0.939	0.998	0.872	0.107	-0.109
fft1	0.960	0.891	0.945	0.999	0.937	0.058	-0.021
fibcall	0.623	0.891	0.718	0.793	0.852	-0.048	-0.119
fir	0.872	0.891	0.997	0.872	0.965	0.076	-1.778
fourierbench	0.109	0.891	0.935	0.111	0.866	0.010	-0.203
insertsort	0.872	0.891	0.957	0.872	0.925	-0.240	-1.364
janne-complex	0.967	0.891	0.966	0.999	0.936	4.462	-0.029
jfdctint	0.668	0.891	0.762	0.993	0.767	2.631	-0.233
lcdnum	0.963	0.891	0.952	0.999	0.936	0.016	-0.022
ludcmp	0.422	0.891	0.704	0.576	0.792	0.015	-0.252
matmult	0.967	0.891	0.965	0.999	0.936	0.346	0.017
minver	0.531	0.891	0.572	0.839	0.868	0.005	-0.112
ndes	0.108	0.891	0.238	0.403	0.778	0.117	-0.454
prime	0.967	0.891	0.987	0.977	0.937	1.234	-0.027
qsort-exam	0.597	0.891	0.621	0.867	0.877	0.015	-0.088
qurt	0.935	0.891	0.950	0.947	0.907	0.003	-0.167
recursion	0.925	0.891	0.919	0.960	0.897	2.331	1.371
select	0.958	0.891	0.951	0.986	0.938	-0.015	-0.116
shadriver	0.971	0.891	0.993	0.973	0.948	0.113	-0.853
sqrt	0.846	0.891	0.966	0.846	0.933	0.019	-0.060
statemate	0.953	0.891	0.930	0.999	0.930	0.073	-0.041

Table 7.10: *Results from original benchmarks, run on a Raspberry PI 4 with PREEMPT_RT Patch application.*

PREEMPT_RT - fibcall

BENCHMARK	ppi	cval	kpss	bds	h	ε	ε pareto
fibcall-original	0.623	0.891	0.718	0.793	0.852	-0.048	-0.119
fibcall-exponential	0.651	0.891	0.940	0.651	0.923	-0.178	-2.731
fibcall-normal	0.952	0.891	0.980	0.940	0.935	0.164	-1.870
fibcall-pareto	0.948	0.891	0.911	1.000	0.933	4.843	-0.084
fibcall-uniform	0.951	0.891	0.917	0.999	0.936	0.042	-0.058
fibcall-weibull	0.955	0.891	0.929	0.999	0.936	0.019	-0.054

Table 7.11: *Results from the modified versions of "fibcall" benchmark, run on a Raspberry PI 4 with PREEMPT_RT Patch application.*

PREEMPT_RT - minver

BENCHMARK	ppi	cval	kpss	bds	h	ε	ε pareto
minver-original	0.531	0.891	0.572	0.839	0.868	0.005	-0.112
minver-exponential	0.971	0.891	0.978	0.999	0.936	0.025	-0.016
minver-normal	0.955	0.891	0.931	0.999	0.936	0.030	-0.014
minver-pareto	0.557	0.891	0.951	0.557	0.947	0.004	-0.426
minver-uniform	0.000	0.891	0.000	0.042	0.317	-0.046	-1.526
minver-weibull	0.970	0.891	0.975	0.999	0.936	0.046	-0.019

Table 7.12: *Results from modified versions of "minver" benchmarks, run on a Raspberry PI 4 with PREEMPT_RT Patch application.*

PREEMPT_RT - qsort

BENCHMARK	ppi	cval	kpss	bds	h	ε	ε pareto
qsort-original	0.597	0.891	0.621	0.867	0.877	0.015	-0.088
qsort-exponential	0.411	0.891	0.435	0.970	0.835	0.000	-0.582
qsort-normal	0.873	0.891	0.873	0.924	0.891	-0.109	-0.562
qsort-pareto	0.862	0.891	0.862	0.983	0.901	0.098	-0.196
qsort-uniform	0.889	0.891	0.889	0.975	0.911	-0.073	-0.199
qsort-weibull	0.960	0.891	0.944	0.999	0.935	0.254	-0.100

Table 7.13: *Results from modified versions of "qsort" benchmarks, run on a Raspberry PI 4 with PREEMPT_RT Patch application.*

PREEMPT_RT - sorting

BENCHMARK	ppi	cval	kpss	bds	h	ε	ε pareto
insertsort-original	0.872	0.891	0.957	0.872	0.925	-0.240	-1.364
insertsort-indep	0.966	0.891	0.995	0.950	0.953	0.009	-0.092
qsort-original	0.597	0.891	0.621	0.867	0.877	0.015	-0.088
qsort-indep	0.730	0.891	0.751	0.953	0.864	0.021	-0.286

Table 7.14: *Results from sorting benchmarks, run on a Raspberry PI 4 with PREEMPT_RT Patch application.*

PREEMPT_RT - TCAS-sort

BENCHMARK	ppi	cval	kpss	bds	h	ε	ε pareto
tcas-sort	0.000	0.891	0.000	0.000	0.309	-0.040	-0.171
tcas-sort-indep	0.016	0.891	0.166	0.074	0.682	0.166	-0.245

Table 7.15: *Results from TCAS-sort benchmarks, run on a Raspberry PI 4 with PREEMPT_RT Patch application.*

PREEMPT_RT - trivial

BENCHMARK	ppi	cval	kpss	bds	h	ε	ε pareto
exponential	0.931	0.891	0.948	0.935	0.911	0.483	-0.003
normal	0.937	0.891	0.965	0.948	0.896	-0.079	-0.373
pareto	0.941	0.891	0.963	0.931	0.928	1.134	0.433
uniform	0.866	0.891	0.946	0.866	0.924	-0.152	-0.292
weibull	0.953	0.891	0.957	0.987	0.913	2.060	1.369

Table 7.16: *Results from trivial benchmarks, run on a Raspberry PI 4 with PREEMPT_RT Patch application.*

PREEMPT_RT - TCAS

BENCHMARK	ppi	cval	kpss	bds	h	ε	ε pareto
tcas-randomval	0.852	0.891	0.956	0.852	0.922	0.127	-0.127
tcas-realval	0.024	0.891	0.794	0.031	0.758	0.042	-0.025

Table 7.17: *Results from "TCAS" benchmark, run on a Raspberry PI 4 with PREEMPT_RT Patch application.*

PREEMPT_RT - timing							
BENCHMARK	**ppi**	**cval**	**kpss**	**bds**	**h**	ε	ε **pareto**
timing-randomval	0.088	0.891	0.114	0.809	0.735	-0.001	-0.846
timing-realval	0.025	0.891	0.390	0.070	0.609	0.004	-0.230

Table 7.18: *Results from audio compression benchmark, run on a Raspberry PI 4 with PREEMPT_RT Patch application.*

CHAPTER 8

Future Works and Conclusions

In this final section, we present the possible future research activities regarding the WCET benchmarks and their application in probabilistic timing analyses. Then, the conclusions of this thesis are drawn.

8.1 Future Works

In this work, we limited the analysis to WCET benchmarks whose source code is publicly available. For this reason, one of the possible future works is to statistically analyse other already existing benchmarks, belonging to different suites than the ones we have considered. Furthermore, new programs to test probabilistic timing analyses (and in particular MBPTA) could be created, starting from the ones already contained in the common benchmark collections, or even by writing them from scratch. Particular importance should be given to real applications and code linked to the industrial world. Further experiments could be performed through the employment of different hardware architectures or setups than the ones shown in this thesis.

More complex techniques for making benchmarks' execution time independent from input values are promising and, consequently, are good candidates for future studies. The body of the so-called *trivial* benchmarks could be enriched with other instructions than the only increment of a variable, that may have a different impact depending on the architecture. Finally, more focus could be given to the statistical hypotheses (described in Section 2.1.3) that have not been considered in this thesis: the MDA and representativity hypotheses. The dataset of observed execution times of all the experiments run in this work has been made public for further analyses[1], including the verification of these two hypotheses.

8.2 Conclusions

This thesis was focused on the probabilistic analysis of WCET benchmarks. We started our study from the programs belonging to already existing suites (in particular *Mälardalen WCET benchmark suite*), statistically characterizing them through their PPI value, and proving that most of them did not respect EVT hypotheses, a condition that is necessary for the correct computation of the pWCET. After that, we moved to the creation of new benchmarks, to form a new suite for probabilistic analyses. The first set was obtained by introducing randomness on certain benchmark parameters (such as the size of matrixes and arrays, or the number of iterations to be performed), according to specific distributions. This was done to test the ability to fulfil the hypotheses according to the change of the benchmark and input parameters. Then, we revised the code of some benchmarks performing the sorting of numerical arrays, with the goal to make their execution time less dependent on the input values. These modifications showed a great improvement in the EVT hypotheses' satisfaction, and they can, consequently, inspire future works to make existing applications compliant with EVT. The last programs presented in this work, instead, were written from scratch. We started with trivial benchmarks – composed of a *for* loop incrementing the value of an integer value each time –, whose number of iterations was determined by sampling from different distributions. This allowed us to test how the specific distributions' impacts on PPI. Finally, we developed some real applications,

[1] URL: `https://doi.org/10.5281/zenodo.4024316`

in particular one performing compression and decompression of audio files, and two derived from different stages of the TCAS algorithm. Thanks to these applications, we showed the limits of probabilistic real-time, when it is applied without proper preliminary considerations, such as the introduction of input-independent code.

All the benchmarks have been tested on different architectures (i.e. a STM32L010RB microcontroller and a Raspberry PI 4), and with different setups (i.e. measuring time with the internal timer and with a logic analyser in the case of STM32 board, and with and without the application of PREEMPT_RT patch in the case of Raspberry PI board). These hardware and software configurations allowed us to explore the behaviour of the statistical metrics on different scenarios.

Appendices

APPENDIX A

List of Benchmarks

In this section, the list of all the benchmarks considered in this thesis is shown, specifying which suite they have been taken from. Also, for each benchmark suite some information is provided.

Benchmark	Type	Suite
adpcm	pseudo-app	Mälardalen
basicmath	synthetic	MiBench
bitcount	synthetic	MiBench
bsort100	synthetic	Mälardalen
cnt	synthetic	Mälardalen
compress	pseudo-app	Mälardalen
coop	synthetic	WTC14
cover	synthetic	Mälardalen
crc	pseudo-app	Mälardalen
duff	synthetic	Mälardalen
edn	pseudo-app	Mälardalen
expint	synthetic	Mälardalen

Benchmark	Type	Suite
exponentialBenchmark	trivial	Custom
fdct	pseudo-app	Mälardalen
fft1	pseudo-app	Mälardalen
fibcall	synthetic	Mälardalen
fibcall-exponential	synthetic	Mälardalen custom
fibcall-normal	synthetic	Mälardalen custom
fibcall-pareto	synthetic	Mälardalen custom
fibcall-uniform	synthetic	Mälardalen custom
fibcall-weibull	synthetic	Mälardalen custom
fir	pseudo-app	Mälardalen
fourierbench	synthetic	MiBench
insertsort	synthetic	Mälardalen
insertsort-indep	pseudo-app	Mälardalen
janne-complex	synthetic	Mälardalen
jfdctint	pseudo-app	Mälardalen
lcdnum	synthetic	Mälardalen
ludcmp	pseudo-app	Mälardalen
matmult	synthetic	Mälardalen
minver	synthetic	Mälardalen
minver-exponential	synthetic	Mälardalen custom
minver-normal	synthetic	Mälardalen custom
minver-pareto	synthetic	Mälardalen custom
minver-uniform	synthetic	Mälardalen custom
minver-weibull	synthetic	Mälardalen custom
ndes	pseudo-app	Mälardalen
normalBenchmark	trivial	Custom
paretoBenchmark	trivial	Custom
prime	synthetic	Mälardalen
qsort-exam	synthetic	Mälardalen
qsort-exponential	synthetic	Mälardalen custom
qsort-indep	synthetic	Mälardalen custom
qsort-normal	synthetic	Mälardalen custom
qsort-pareto	synthetic	Mälardalen custom
qsort-uniform	synthetic	Mälardalen custom

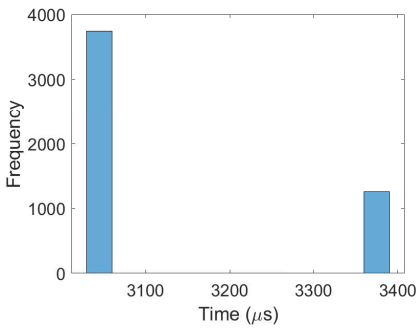
Benchmark	Type	Suite
qsort-weibull	synthetic	Mälardalen custom
qurt	synthetic	Mälardalen
recursion	synthetic	Mälardalen
select	synthetic	Mälardalen
shadriver	synthetic	MiBench
sqrt	synthetic	Mälardalen
statemate	real-app	Mälardalen
tcas-randomvalues	real-app	Custom
tcas-realvalues	real-app	Custom
tcas-sort	real-app	Custom
tcas-sort-indep	real-app	Custom
timing-randomval	real-app	Mediabench custom
timing-realval	real-app	Mediabench custom
ud	pseudo-app	Mälardalen
uniformBenchmark	trivial	Custom
weibullBenchmark	trivial	Custom

Suite	Link	License
Mälardalen [16]	http://www.mrtc.mdh.se/projects/wcet/benchmarks.html	GPL
MediaBench [20]	https://cs.slu.edu/~fritts/mediabench/	LLVM
MiBench [23]	http://vhosts.eecs.umich.edu/mibench//index.html	FREE with no restrictions
WTC'14	https://github.com/t-crest/patmos-benchmarks/tree/master/WTC14-misc	-

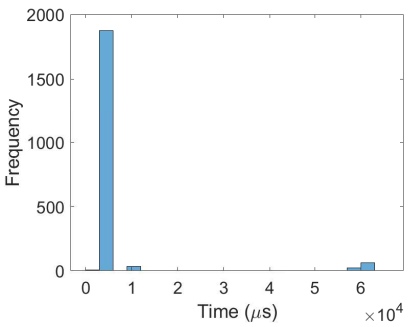
APPENDIX B

Histograms

In this section, histograms showing the distribution of the execution times of all the benchmarks discussed in this Thesis are shown. In particular, the plots refer to the measurements deriving from the employment of the internal timer of the STM32 board.

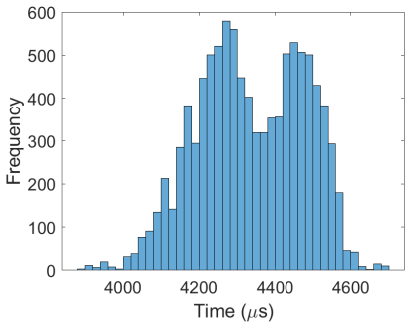


adpcm

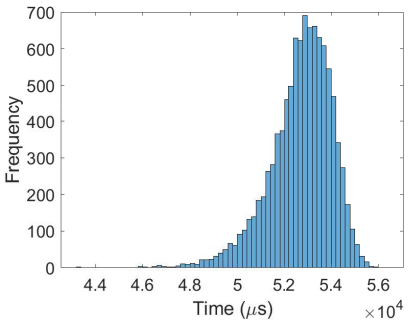


basicmath

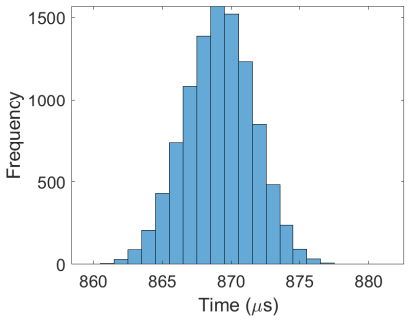
Appendix B. Histograms



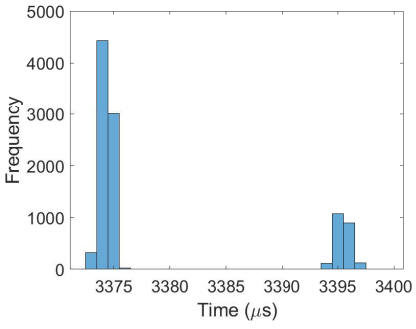
bitcount



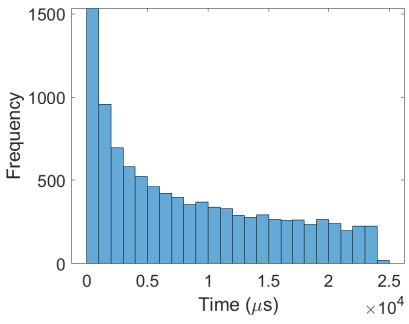
bsort100



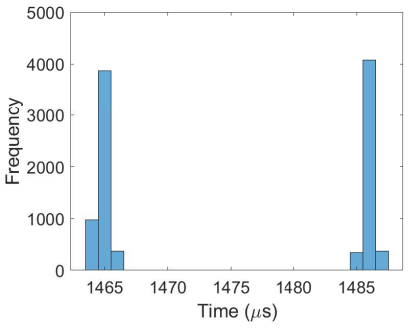
cnt



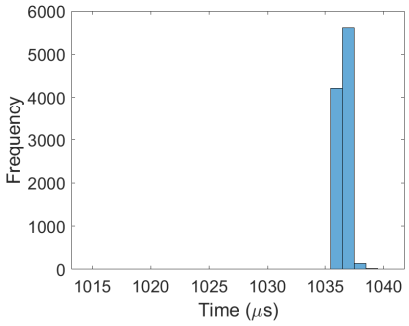
compress



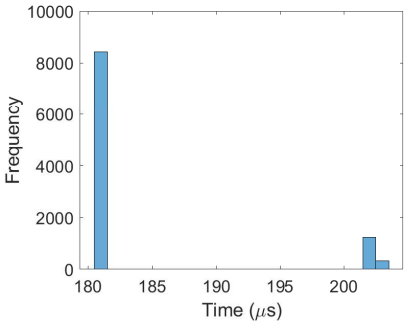
coop



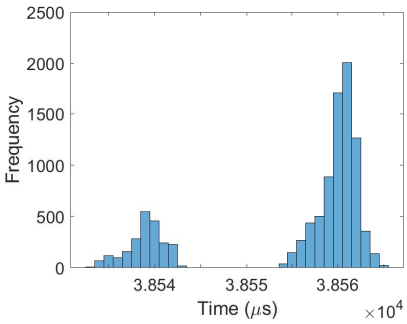
cover



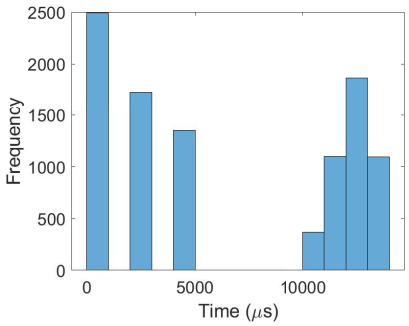
crc



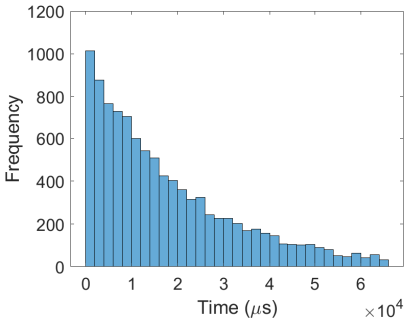
duff



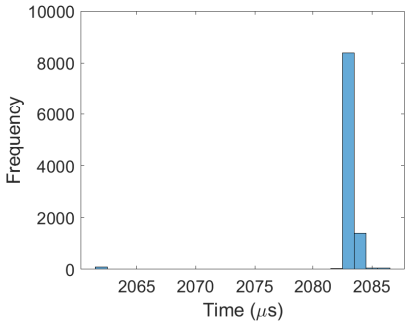
edn



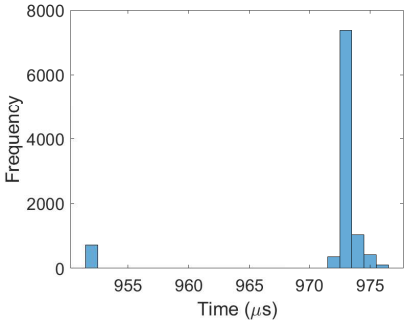
expint



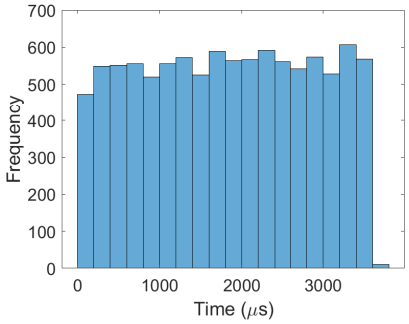
exponentialBenchmark



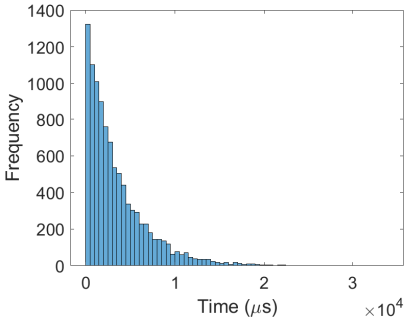
fdct



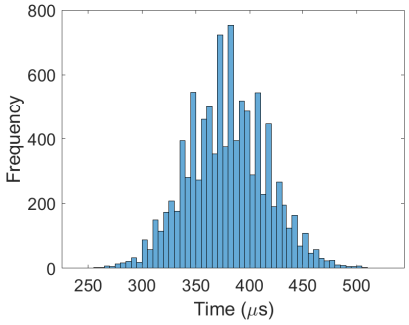
fft1



fibcall

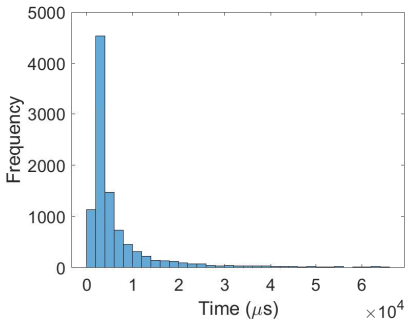


fibcall-exponential

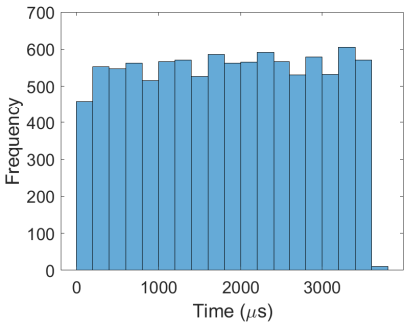


fibcall-normal

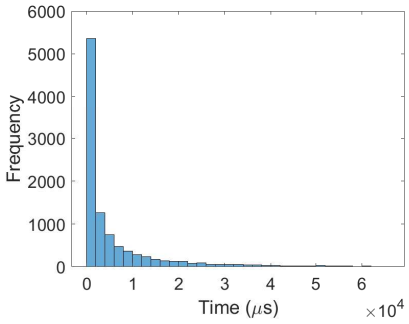
Appendix B. Histograms



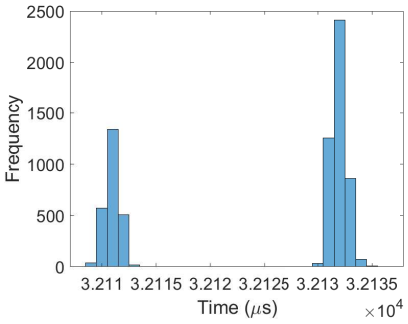
fibcall-pareto



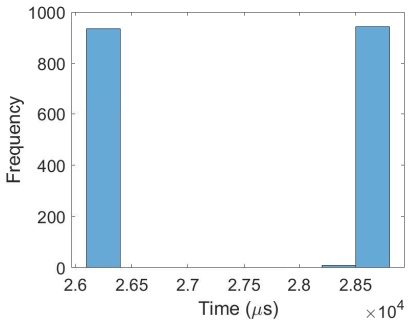
fibcall-uniform



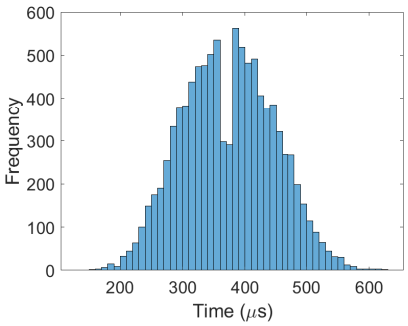
fibcall-weibull



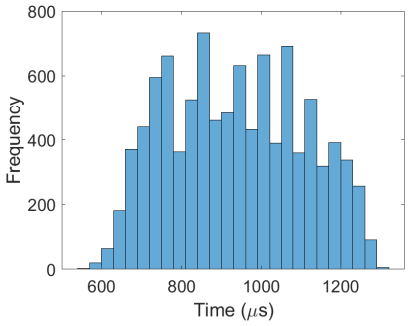
fir



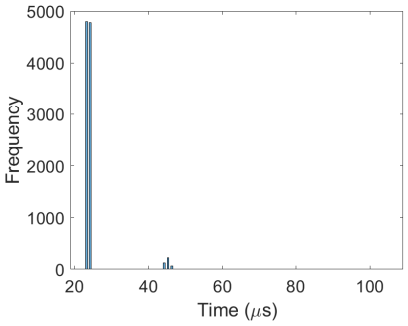
fourierbench



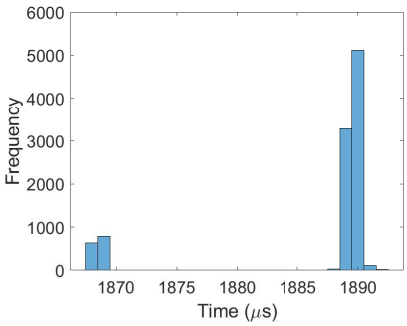
insertsort



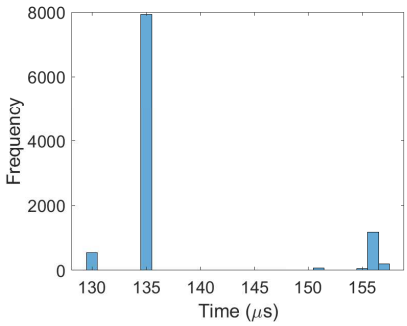
insertsort-indep



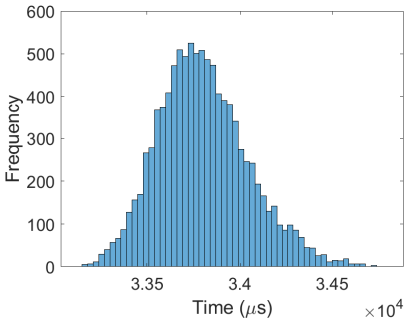
janne-complex



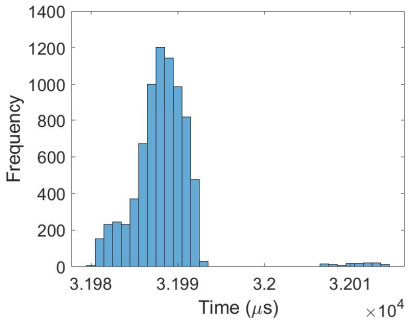
jfdctint



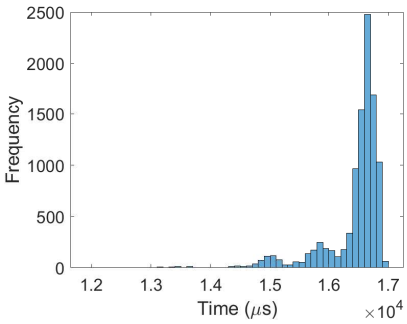
lcdnum



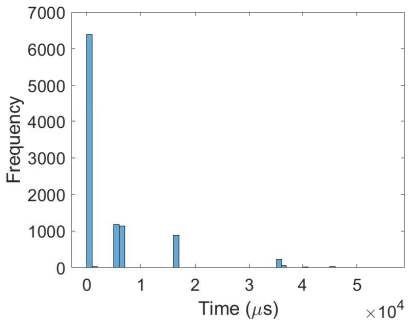
ludcmp



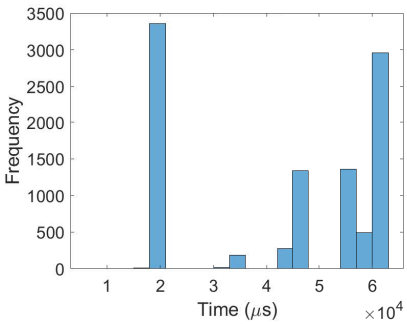
matmult



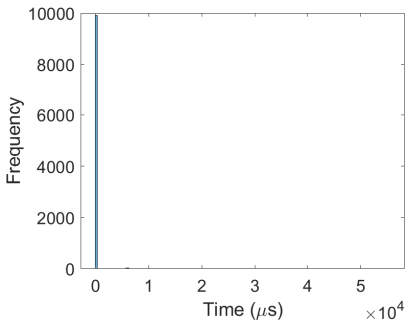
minver



minver-exponential

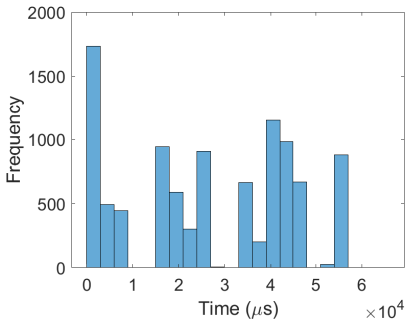


minver-normal

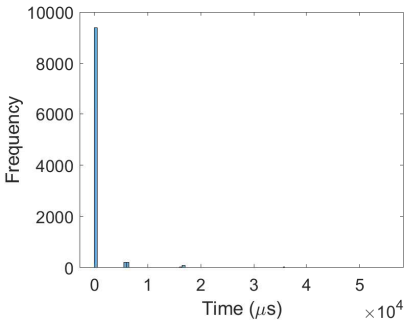


minver-pareto

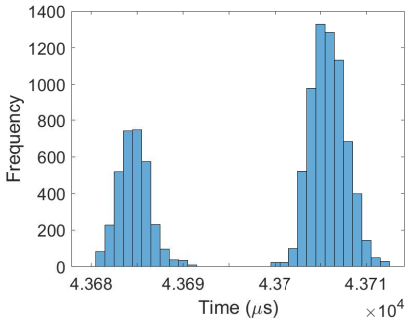
Appendix B. Histograms



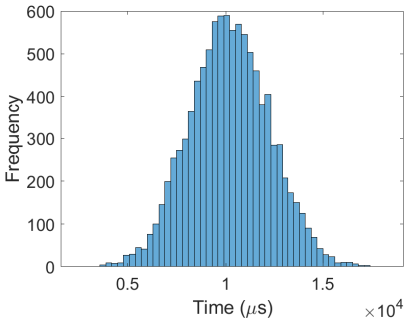
minver-uniform



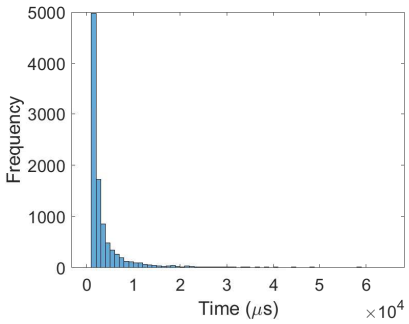
minver-weibull



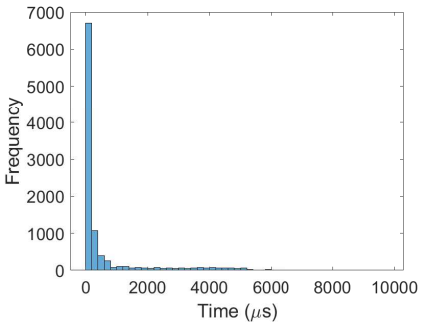
ndes



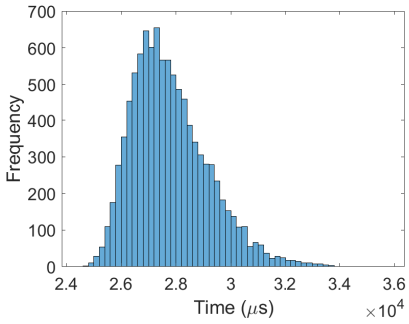
normalBenchmark



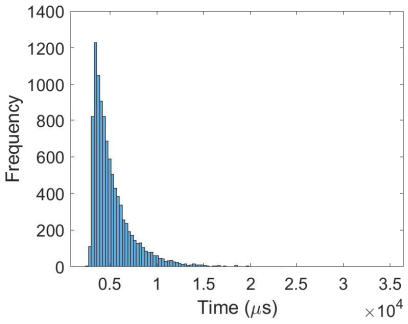
paretoBenchmark



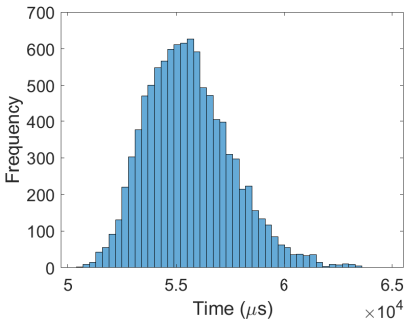
prime



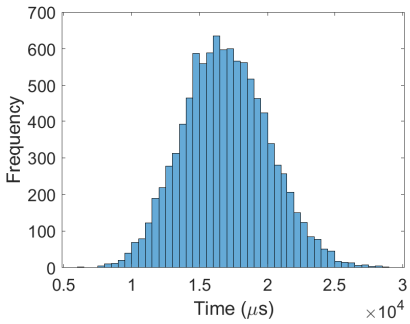
qsort-exam



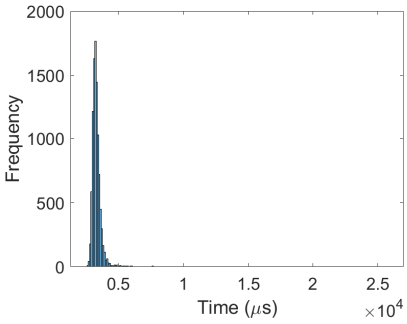
qsort-exponential



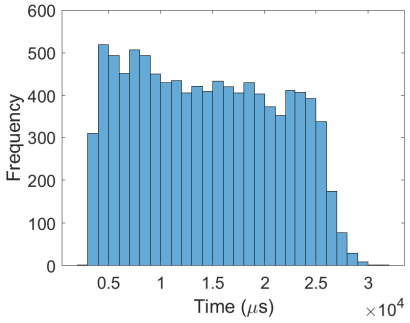
qsort-indep



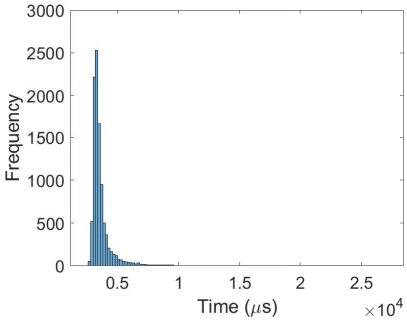
qsort-normal



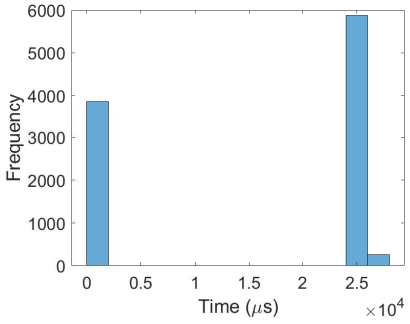
qsort-pareto



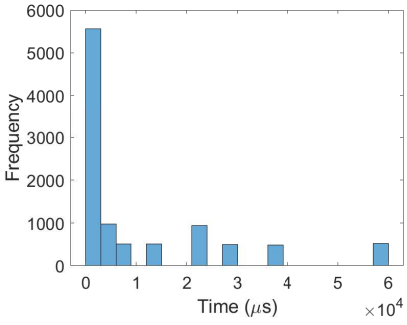
qsort-uniform



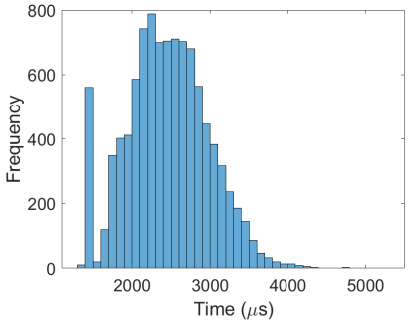
qsort-weibull



qurt

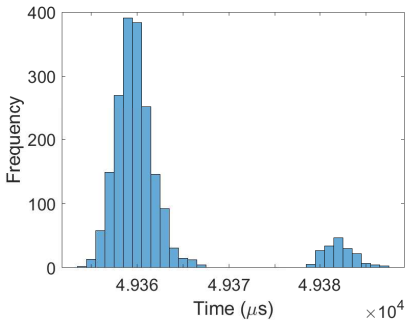


recursion

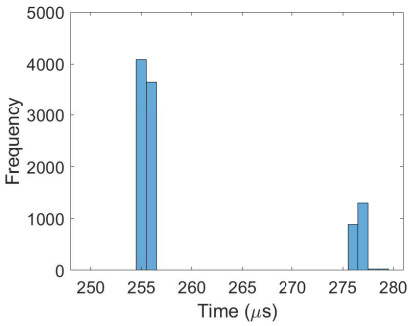


select

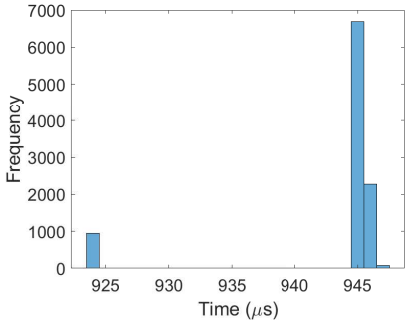
Appendix B. Histograms



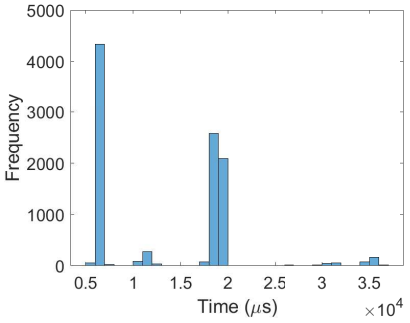
shadriver



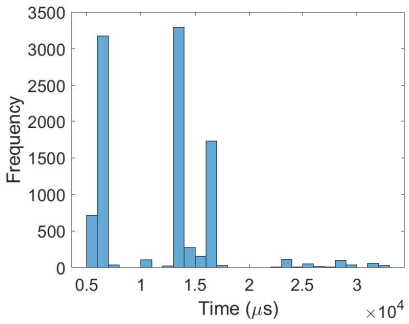
sqrt



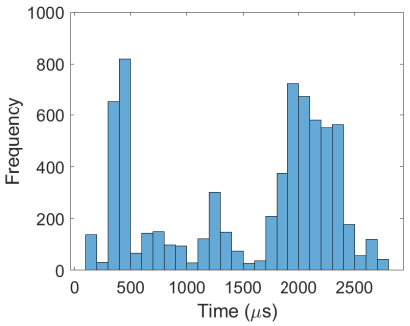
statemate



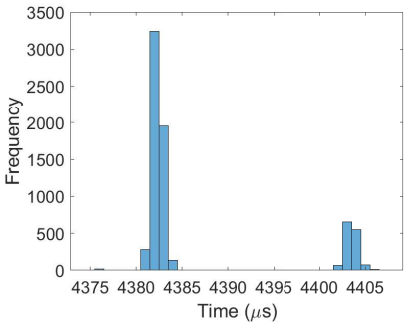
tcas-randomval



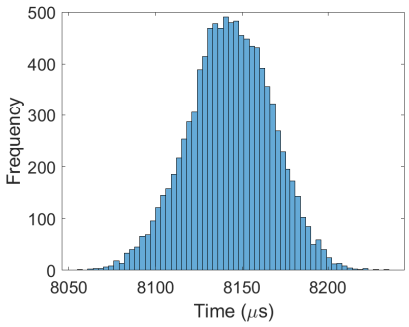
tcas-realvalues



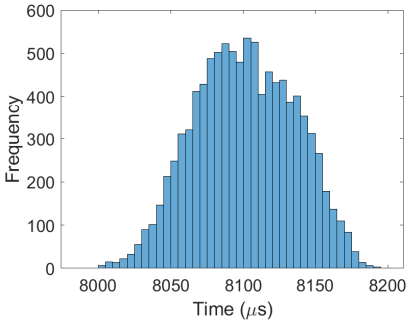
tcas-sort



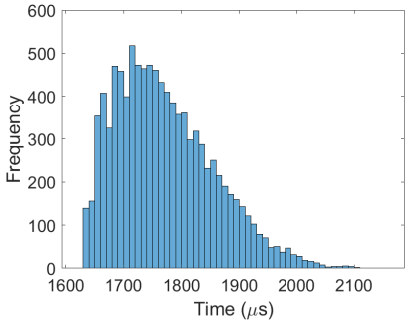
tcas-sort-indep



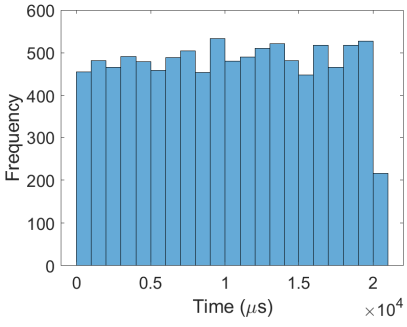
timing-randomval



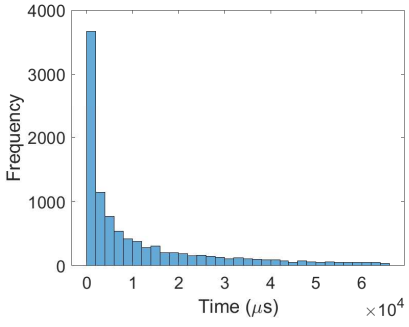
timing-realval



ud



uniformBanchmark



weibullBenchmark

Bibliography

[1] J. Abella, D. Hardy, I. Puaut, E. Quiñones, and F. J. Cazorla. On the comparison of deterministic and probabilistic wcet estimation techniques. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 266–275, 2014.

[2] Jaume Abella, Maria Padilla, Joan Del Castillo, and Francisco J. Cazorla. Measurement-based worst-case execution time estimation using the coefficient of variation. *ACM Trans. Des. Autom. Electron. Syst.*, 22(4), June 2017.

[3] Kostiantyn Berezovskyi, Luca Santinelli, Konstantinos Bletsas, and Eduardo Tovar. Wcet measurement-based and extreme value theory characterisation of cuda kernels. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 279–288, New York, NY, USA, 2014. Association for Computing Machinery.

[4] A. Burns and S. Edgar. Predicting computation time for advanced processor architectures. In *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*, pages 89–96, 2000.

[5] 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 *2012 24th Euromicro Conference on Real-Time Systems*, pages 91–101, 2012.

[6] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[7] S. Edgar and A. Burns. Statistical analysis of wcet for scheduling. In *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420)*, pages 215–224, 2001.

[8] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Sørensen, Peter Wägemann, and Simon Wegener. Taclebench: a benchmark collection to support worst-case execution time research. 01 2016.

[9] Ronald Aylmer Fisher and Leonard Henry Caleb Tippett. Limiting forms of the frequency distribution of the largest or smallest member of a sample. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 24, pages 180–190. Cambridge University Press, 1928.

[10] William Fornaciari, Giovanni Agosta, David Atienza, Carlo Brandolese, Leila Cammoun, Luca Cremona, Alessandro Cilardo, Albert Farres, José Flich, Carles Hernandez, Michal Kulchewski, Simone Libutti, José Maria Martínez, Giuseppe Massari, Ariel Oleksiak, Anna Pupykina, Federico Reghenzani, Rafael Tornero, Michele Zanella, Marina Zapater, and Davide Zoni. Reliable power and time-constraints-aware predictive management of heterogeneous exascale systems. In *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '18, pages 187–194, New York, NY, USA, 2018. Association for Computing Machinery.

[11] Matteo Fusi, Fabio Mazzocchetti, Albert Farres, Leonidas Kosmidis, Ramon Canal, Francisco J. Cazorla, and Jaume Abella. On the use of probabilistic worst-case execution time estimation for parallel applications in high performance systems. *Mathematics*, 8(3):314, Mar 2020.

[12] Lima G. and Bate. I. Valid application of evt in timing analysis by randomising execution time measurements. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[13] Boris Vladimirovich Gnedenko. On a local limit theorem of the theory of probability. *Uspekhi Matematicheskikh Nauk*, 3(3):187–194, 1948.

[14] David Griffin and Alan Burns. Realism in Statistical Analysis of Worst Case Execution Times. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 44–53, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7.

[15] Fabrice Guet, Luca Santinelli, and Jérôme Morio. On the Reliability of the Probabilistic Worst-Case Execution Time Estimates. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, January 2016.

[16] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks: Past, present and future. volume 15, pages 136–146, 01 2010.

[17] J. Hansen, S. A. Hissam, and G. A. Moreno. Statistical-based wcet estimation and validation. In *Proceedings of the Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 252, 2009.

[18] M Ross Leadbetter, Georg Lindgren, and Holger Rootzén. Conditions for the convergence in distribution of maxima of stationary normal processes. *Stochastic Processes and their Applications*, 8(2):131–139, 1978.

[19] Pierre L'Ecuyer. Tables of maximally equidistributed combined lfsr generators. *Mathematics of computation*, 68(225):261–269, 1999.

[20] Chunho Lee, Miodrag Potkonjak, and William Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. pages 330–335, 12 1997.

[21] G. Lima and I. Bate. Valid application of evt in timing analysis by randomising execution time measurements. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 187–198, 2017.

[22] G. Lima, D. Dias, and E. Barros. Extreme value theory for estimating task execution time bounds: A careful look. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.

[23] Guthaus Matthew R., Ringenberg Jeffrey S., Ernst Dan, Austin Todd M., Mudge Trevor, and Brown Richard B. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, 12 2001.

[24] C. Maxim, A. Gogonel, I. Asavoae, M. Asavoae, and L. Cucu-Grosjean. Reproducibility and representativity: Mandatory properties for the compositionality of measurement-based wcet estimation approaches. *SIGBED Rev.*, 14(3):24–31, November 2017.

[25] César Munoz, Anthony Narkawicz, and James Chamberlain. A tcas-ii resolution advisory detection algorithm. In *AIAA Guidance, Navigation, and Control (GNC) Conference*, page 4622, 2013.

[26] Lara Premi, Federico Reghenzani, Giuseppe Massari, and William Fornaciari. A game theory approach to heterogeneous resource management. In *Proceedings of the International Conference on Embedded Software Companion*, EMSOFT '20. Association for Computing Machinery, 2020.

[27] F. Reghenzani, G. Massari, and W. Fornaciari. The misconception of exponential tail upper-bounding in probabilistic real time. *IEEE Embedded Systems Letters*, 11(3):77–80, 2019.

[28] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. The real-time linux kernel: A survey on preempt_rt. *ACM Comput. Surv.*, 52(1), February 2019.

[29] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. Probabilistic-wcet reliability: Statistical testing of evt hypotheses. *Microprocessors and Microsystems*, 77:103135, 2020.

[30] Federico Reghenzani, Giuseppe Massari, William Fornaciari, and Andrea Galimberti. Probabilistic-wcet reliability: On the experimental validation of evt hypotheses. In *Proceedings of the International Conference on Omni-Layer Intelligent Systems*, COINS '19, pages 229–234, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Federico Reghenzani, Luca Santinelli, and William Fornaciari. Why statistical power matters for probabilistic real-time: Work-in-progress. In *Proceedings of the International Conference on Embedded Software Companion*, EMSOFT '19, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Federico Reghenzani, Luca Santinelli, and William Fornaciari. Dealing with uncertainty in pWCET estimations. *ACM Trans. Embed. Comput. Syst.*, 19(5), 2020.

[33] Luca Santinelli and Zhishan Guo. On the criticality of probabilistic worst-case execution time models. In Kim Guldstrand Larsen, Oleg Sokolsky, and Ji Wang, editors, *Dependable Software Engineering. Theories, Tools, and Applications*, pages 59–74, Cham, 2017. Springer International Publishing.

[34] Luca Santinelli, Jérôme Morio, Guillaume Dufour, and Damien Jacquemart. On the Sustainability of the Extreme Value Theory for WCET Estimation. In Heiko Falk, editor, *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 21–30, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.