UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH
BARCELONA SCHOOL OF INFORMATICS

CPU Performance Signatures for Security Attacks Detection

Author Miquel Baena Sanfeliu

July 4th, 2018

Director: Carles Hernández (Department of Computer Architecture)

Co-director: Leonidas Kosmidis (Barcelona Supercomputing Center)

Ponent: Miquel Moretó Planas (Department of Computer Architecture)

Bachelor Degree in Informatics Engineering

Specialization: INFORMATIONS TECHNOLOGY

FACULTAT D'INFORMATICA DE BARCELONA (FIB)

UNIVERSITAT POLITÉCNICA DE CATALUNYA (UPC) - BarcelonaTech

Acknowledgement

I would like to thank my project directors, Dr. Carles Hernández and Leonidas Kosmidis for guiding and supporting me over this five months. You have set an example of excellence as a researcher, mentor, instructor, and role model.

I would also like to thank to Barcelona Computer Center for giving me resources like the cluster use, and also to Architecture Department of UPC for giving me a remote machine use.

Finally, I would like to thank and dedicate this thesis to my parents Miguel Baena and Mª Teresa Sanfeliu and my sister Elisabet Baena for helping me every day in everything.

Table of content

Index of Illustrations

Index of tables

Index of plots

Abstract

A new approach for detecting security attacks on real-time embedded applications by using performance signatures is introduced in the thesis. Assuming that the behavior of real-time embedded applications is quite stable i.e suffers from little variability, using performance signatures, or in other words key performance metric estimations of the applications, arises as a good option to detect the potential malfunctioning of the system caused by the presence of malware or by the simple existence of bugs escaping the verification process. In our approach we also take into account that slight modifications in the memory layout can lead to high deviations in the performance of the applications for CPUs using standard cache designs implementing modulo placement and LRU replacement. This performance variability may jeopardize the utilization of performance signatures on top of these processors. Therefore, we will also analyze the suitability of performance signatures in the context of time-randomized processors since these processors have been shown to provide higher performance stability.

The underlying assumption of this work is that significant performance deviations from well-behaved systems can be used to trigger alerts about the presence of security attacks. In order to be able to study the behavior of the CPU performance in different cases, we will use a SPARC simulator resembling the NGMP (Next Generation Microprocessor) processor, a source-to-source compiler called TASA to mimic the variability due to modifications in the cache layout. We also use the EEBMC Autobench benchmarks as representative workloads of the automotive industry. From these benchmarks we have generated randomized variants with TASA to simulate the cache layout variability. We have observed that most of the cases performance signatures suffices to detect the presence of malware since intrinsic cache variability is reduced.

1 Introduction

This project is a TFG (Treball de Final de Grau) developed in FIB (Facultat d'Informatica de Barcelona) in collaboration with the BSC (Barcelona Supercomputing Center). This project aims to study to what extent using CPU performance signatures for applications executed in the processor can be used to detect the presence of malware or Security Attacks. In particular, in this TFG we focus on the utilization of performance signatures in the context of real-time safety-related applications like those employed in cars.

Nowadays, the CPUs are ubiquitous in electronic systems and most of the technological products include a CPU. We can find CPUs in a wide range of devices from supercomputers, to satellites and airplanes, and smart transportation systems in general. When CPUs control critical applications the security of the systems becomes of paramount importance forcing system designers to consider security aspects as a primary design concern. While safety-critical systems are subject to a thorough verification process to validate the behavior and correctness of these systems there are a number of aspects that challenge the security of these systems. First, safety-critical software is often developed by third parties companies and it is the responsibility of the system integrator to validate the correctness of the final system. While the third parties are expected to be trustable the fact that the software development process is not fully controlled by the system integrator forces system designers to minimize the existence of potential security flaws. Second, with the advent of new applications like autonomous driving systems the need for having access to the internet of these devices grows as a way to be able to receive software updates. Interconnecting safety-critical systems either to private or public networks opens the door to potential security attacks to those systems.

In the context of real-time systems, the performance of the applications running in CPUs is well optimized and normally average performance of applications running on top of general purpose processors is a good metric to understand the behavior of the system. This is so because CPUs are adapted to take advantage of programs properties like temporal and spatial locality and include hardware features like caches and branch predictors able to exploit such properties. In the context, of embedded systems the behavior of applications is even easier to predict since programs are generally small (fit in cache) and processors are generally very simple [1].

Assuming that the behavior of real-time embedded applications is quite stable i.e suffers from little variability using performance signatures, or in other words key performance metric estimations of the applications, arises as a good option to detect the potential malfunctioning of the system caused by the presence of malware or by the simple existence of bugs escaping the verification process.

However, in spite of embedded CPUs today are able to run most of the programs at very good and so predictable performance, there are cases where program runs have a very bad performance (a.k.a pathological cases). Performance deviations can occur in embedded systems when executing cache sensitive software after slight memory layout modifications. Memory layout changes can occur when any piece of software in the system is modified which can be caused by a software update of the system software.

What we are looking for in this project is to be able to understand up to which extent we can use performance signatures to distinguish between expected CPU performance variability caused by changes in the memory layout, as one of the main factors affecting performance predictability in real-time embedded systems, and the performance of the CPU in the presence of malicious software by using CPU performance signatures. As mentioned before, memory layout modifications can introduce significant performance deviations in the performance of applications and thus, it is difficult to understand whether performance variability is harmful (due to malicious software or the presence of bugs) or innocuous. Our hypothesis is that slight modifications in the memory layout can lead to high deviations in the performance of the applications for CPUs using standard cache designs implementing modulo placement and LRU replacement. This may jeopardize the utilization of performance signatures on top of these processors. Therefore, we will also analyze the suitability of performance signatures in the context of time-randomized processors [2] since these processors have been shown to provide higher performance stability.

In order to be able to study the behavior of the CPU performance in different cases, we will use a SPARC simulator resembling the NGMP (Next Generation Microprocessor) processor [3] that was developed by Cobham Gaisler and European Space Agency, a source-to-source compiler called TASA [4] and the EEBMC Autobench benchmarks [5] as representative workloads of the automotive industry.

1.1 Summary of the contents

In Section 2 we review the background of this thesis.

Section 3 describes how to build a mechanism to detect malware based on the utilization of performance signatures.

Section 4 describes the methodology how to get a performance signature from some workloads.

Section 5 presents the evaluation and the obtained results.

Finally, Section 6 shows the conclusions obtained from the evaluation of the obtained results.

2 Background

This project analyzes the suitability of using CPU performance signatures for Security Attacks detection. Due to the short time we have to develop the project, we only will be able to perform the study for a family of CPUs based on the SPARC ISA. In particular we will use the NGMP [3] as the reference CPU in this work.

There are several studies that have analyzed the problem of analyzing the presence of malware in computers. For instance, in [6] authors characterize the performance deviations caused by the presence of malware to understand if these deviations offer some special behavior. A hardware assisted mechanism for the detection of malware and software vulnerabilities was proposed in [7].

From a different perspective many works have tried to characterize memory-layout induced performance variability. Authors in [8] proposed the Stabilizer tool that performs memory layout randomization dynamically at runtime, to be able to characterize performance of applications without the performance noise introduced in a system. Similarly, the use of randomization techniques has been proposed in [9] to understand worst-case impact of cache variability in the execution time of applications. To do so, authors proposed a source-to-source compiler to implement layout randomization at the source code level.

2.1 Cache variability

In this project we will take into account the impact of caches in performance variability in order to be able to create performance signatures to detect security attacks in real time. Understanding cache variability consists on knowing the normal behavior of the CPU. Characterizing regular cache variability requires collecting different statistics from the execution of the same program. In particular, we will obtain a different number of accesses to cache, number of misses, hit/miss ratio, etc.

This project aims to study which will be the normal working of a cache for some EEMBC Autobench benchmarks and take into account all the small variabilities that are given in cache for good executions of a program (adding the pathological cases), in order to be able to differentiate them from the major number of variabilities that will produce a program that contains malicious code.

2.2 Cache implementations

In this thesis we will consider two different implementations of cache defined by the placement and replacement polices:

- **Commercial off-the-shelf (COTS):** we will be using LRU as a placement policy and modulo as replacement policy.
- **Time-randomized cache:** we will be using random eviction as a replacement policy and enhanced random modulo as placement policy.

COTS cache implementation are the traditional implementations for caches. In this implementation we can use different replacement and placement algorithms but we have chosen modulo and LRU as the most used implementations for placement and replacement, respectively. One of the advantages of this cache policies is that in optimal cases they have a very good performance. The other side of the coin is that for some particular cases (a.k.a pathological cases) these policies provide very bad performance.

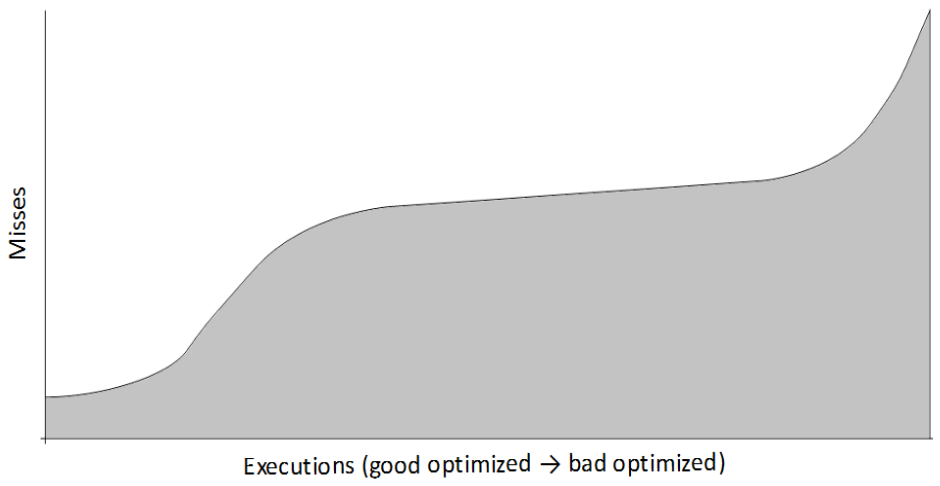


Illustration 1: Example of the behavior of a cache with COTS implementation

Time-randomized cache implementation have been proposed to be able to characterize the cache conflicts probabilistically [10]. They consists on placing and replacing the data in the cache using randomized algorithms from memory direction, in this way we make sure that in bad optimized cases the accesses to cache are the different for each execution. One of its advantages will be that in all cases we will have a very similar performance so in bad cases we will have “normal” performance, but in optimal cases we will have worst performance than COTS implementation. In this project we will call the refer to randomized caches as the one implemented in Time randomized processor [1].

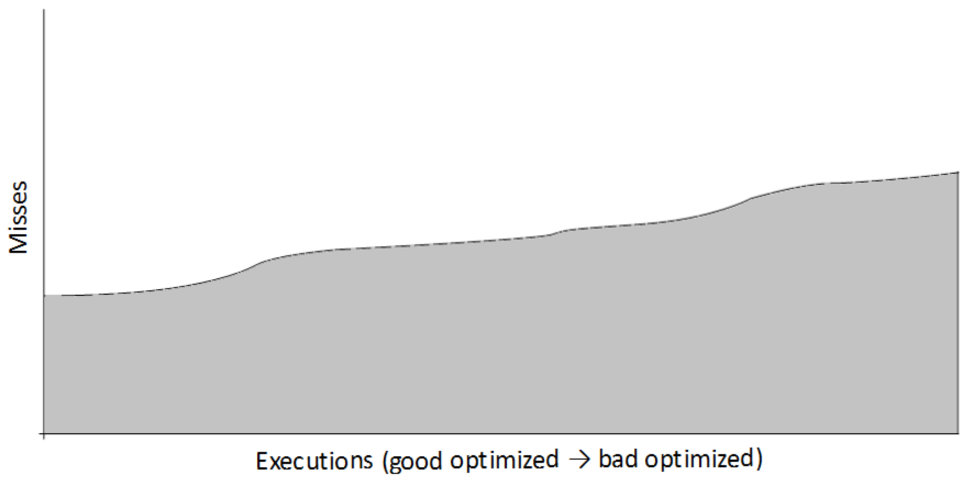


Illustration 2: Example of the behavior of a cache with MBPTA implementation

2.3 Placement and replacement polices

The placement policies determine where a particular memory block can be placed when it goes into the cache while the replacement polices is the algorithm that must choose which items to discard to make room for the new ones when the cache is full.

2.3.1 COTS Cache polices

- **LRU (Last Recently Used):** is a replacement policy that discards the least recently used items first. This algorithm requires keeping track of what was used when, which is expensive if one wants to make sure the algorithm always discards the least recently used item.

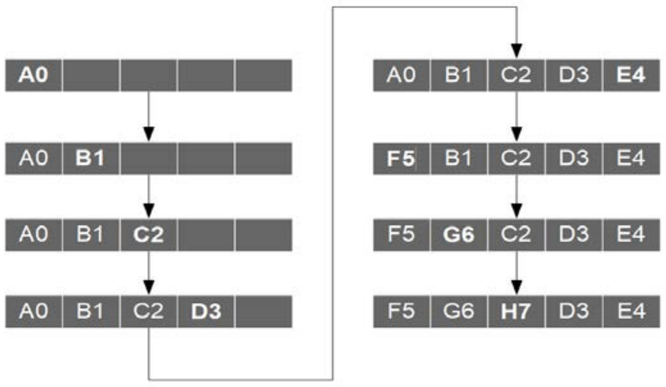


Illustration 3: Example of LRU replacement

- **Modulo (set-associate):** This is the placement policy used in traditional cache architectures, we can find it illustrated in Figure 5. A memory line at address L is placed in, or mapped to, the cache set L mod S where there are S sets in the cache. The memory line may be placed within any cache line in the set, i.e., placement is fully associative within the set. We will refer to the memory lines mapped to a cache set as a conflict set and the preceding placement policy will be referred to as modulo placement. Thus, S contiguous lines in memory are mapped to S distinct sets in the cache and all conflict sets have equal cardinality.

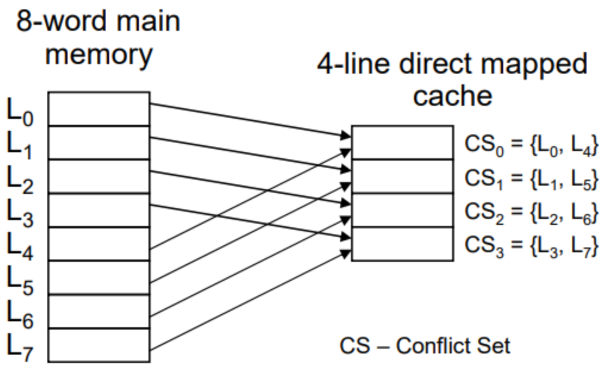


Illustration 4: Example of modulo placement

2.3.2 Time-randomized cache polices

- **Random eviction on miss:** It randomly selects a candidate element and discards it to make room when it occurs a miss. This algorithm does not require maintaining any information about the access history.

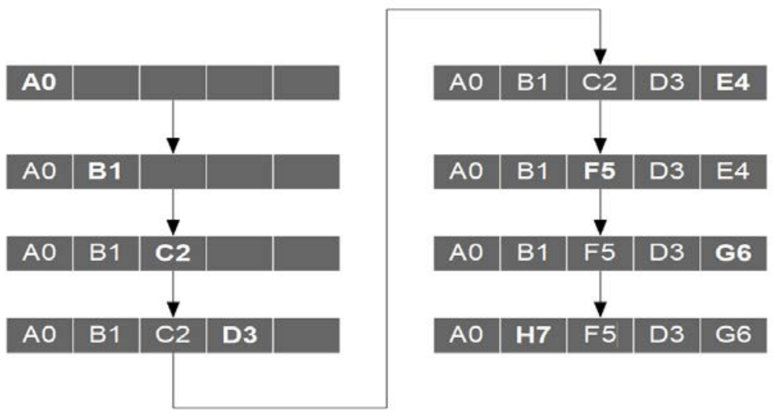
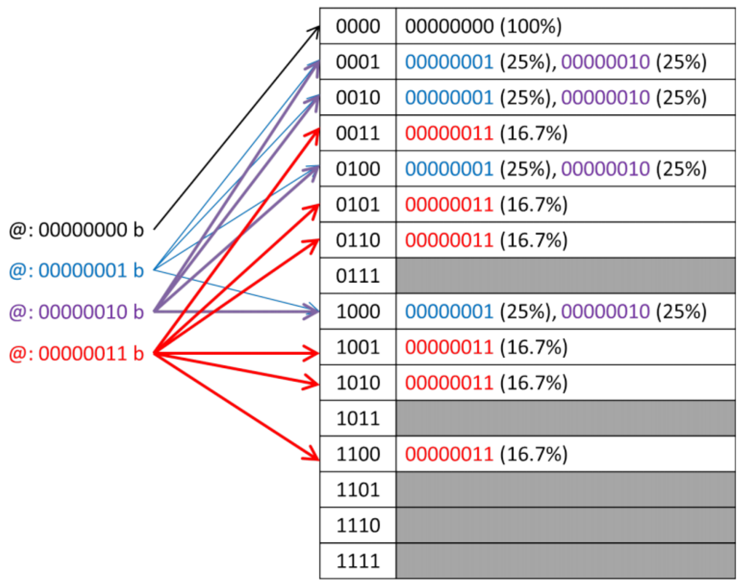


Illustration 5: Example of random replacement

- **Random on run:** Random placement maps each cache line randomly in a cache set based on a random seed that is changed across program runs. While this allows to capture cache conflicts probabilistically, it may produce bad placements in terms of miss rates: even with programs accessing few cache lines, when those lines are randomly placed in the same cache set. Conversely, deterministic modulo placement maps consecutive memory lines into consecutive sets, thus avoiding this type of conflicts. However, conflicts across lines are not random and strictly depend on memory location.



0000	00000000 (100%)
0001	00000001 (25%), 00000010 (25%)
0010	00000001 (25%), 00000010 (25%)
0011	00000011 (16.7%)
0100	00000001 (25%), 00000010 (25%)
0101	00000011 (16.7%)
0110	00000011 (16.7%)
0111	
1000	00000001 (25%), 00000010 (25%)
1001	00000011 (16.7%)
1010	00000011 (16.7%)
1011	
1100	00000011 (16.7%)
1101	
1110	
1111	

Illustration 6: Example of distribution for a 16-set cache with random on run design

- **Enhanced random modulo:** Enhanced random placement aims at obtaining the randomization properties of Hash-Based Random Placement (hRP) while exploiting spatial locality as the modulo placement function does. RM exploits spatial locality by removing conflicts among addresses belonging to the same cache segmentRM creates a randomization of the index bits such that (in every run) with a seed seedi, A is mapped to any (random) set lA = setseedi rm (A) and B to lB = setseedi rm (B) and lA and lB are necessarily different: setmod(A) 6= setmod(B) ∧ bA/CWbc = bB/CWbc → setseedi rm (A) 6= setseedi rm (B) ∀ seedi RM makes a random permutation of the address index bits that is driven by a combination of a random seed (changed across runs) and the upper bits of the address. This removes the dependence among memory mapping and cache layout, and further ensures that the index permutation covers probabilistically cache conflicts. Moreover, unlike hRP, RM exploits spatial locality, improving average performance and pWCET estimates w.r.t. hRP.

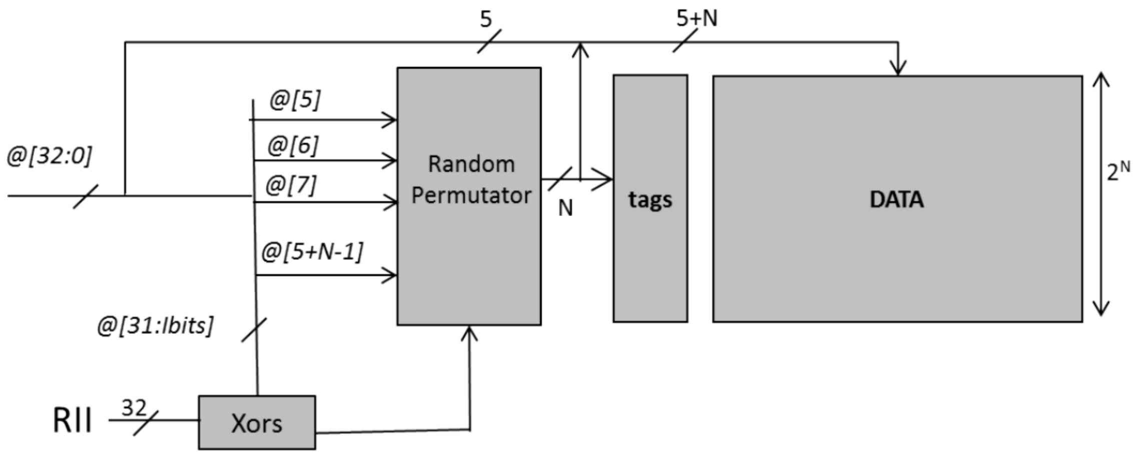


Illustration 7: Random modulo implementation

In the illustrations 13 and 14 we will be able to see the implementation differences between the random on run and the enhanced random modulo.

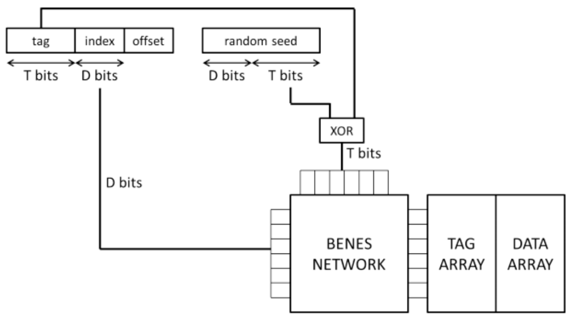


Illustration 8: Schematic of the baseline implementation of a random modulo cache

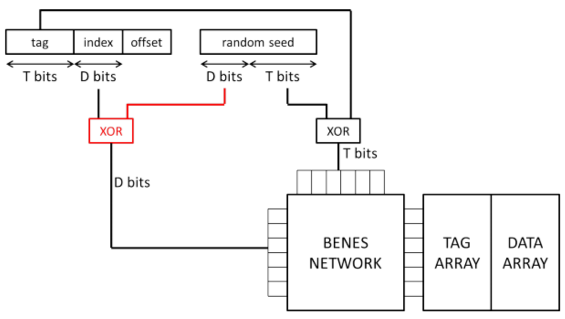


Illustration 9: Schematic of the enhanced implementation of a random modulo

Pathological cases: In COTS cache implementations like LRU and modulo pathological cases leading to very bad performance can occur while the pathological cases of time-randomized cache are less severe. For instance, in LRU with a 2-set associative cache where addresses A and B are mapped to the same set the sequence of access A, B, C, A, B, C, A, B, C. A, B, C will have a 100% miss rate. The reason for this is that LRU evicts the LRU cache line and for this particular sequence the evicted line matches the one that will be accessed next. The same sequence of access in the context of a time-randomized cache will have a better hit rate since the line to be accessed next will not be systematically evicted before.

2.4 Interconnected Devices and Software updates (e.g CARS)

Studies on autonomous cars have been present since the year 1920. But it is not until 2000 when cars start including software. The fact that cars are subject to constant development and improvement will make possible to have up-to-date cars if software updates are allowed. Software updates will bring not only the possibility to achieve higher driving efficiency but will also introduce a high risk for the security of cars since this will open the door to the introduction of malware. At the same, time these software updated will modify the memory layout and cause performance deviations that have to be considered to ensure the real-time operation of the system.

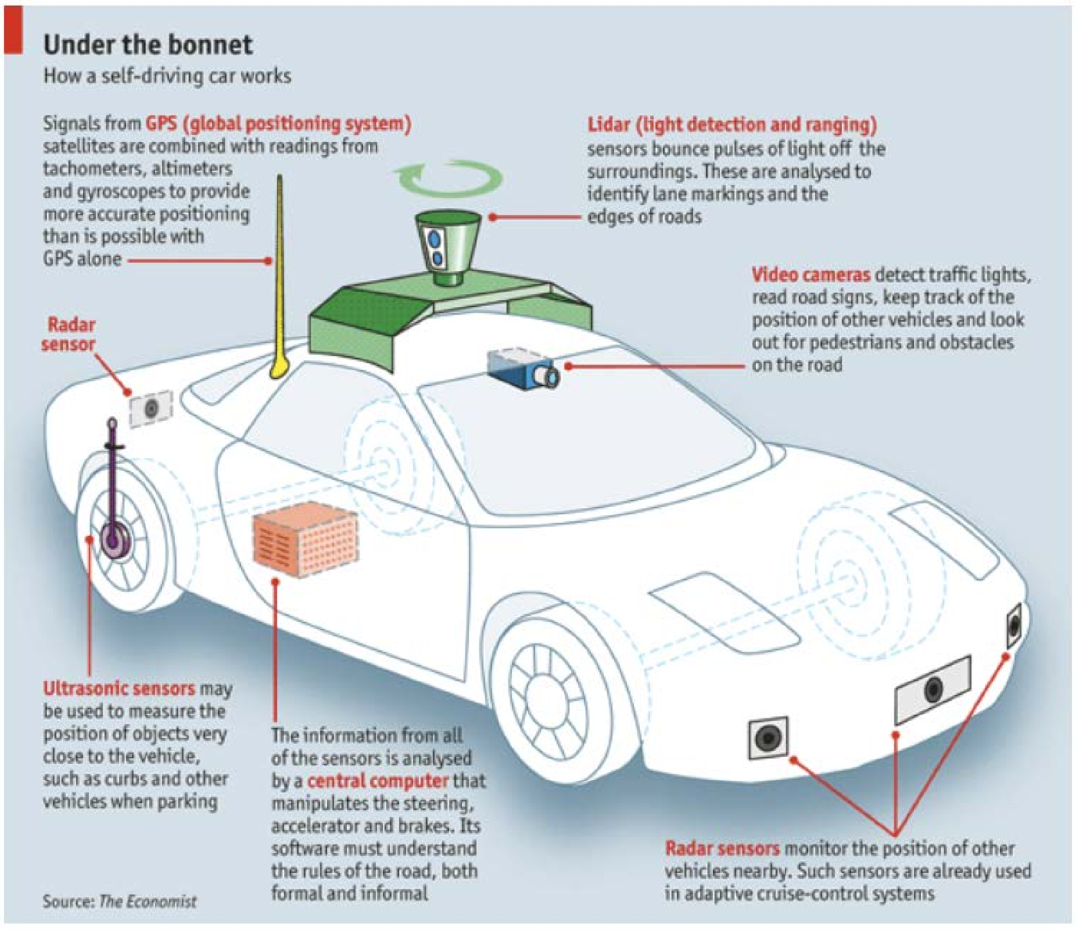

Under the bonnet
How a self-driving car works
Signals from GPS (global positioning system) satellites are combined with readings from tachometers, altimeters and gyroscopes to provide more accurate positioning than is possible with GPS alone
Lidar (light detection and ranging) sensors bounce pulses of light off the surroundings. These are analysed to identify lane markings and the edges of roads
Video cameras detect traffic lights, read road signs, keep track of the position of other vehicles and look out for pedestrians and obstacles on the road
Radar sensor
Ultrasonic sensors may be used to measure the position of objects very close to the vehicle, such as curbs and other vehicles when parking
The information from all of the sensors is analysed by a central computer that manipulates the steering, accelerator and brakes. Its software must understand the rules of the road, both formal and informal
Radar sensors monitor the position of other vehicles nearby. Such sensors are already used in adaptive cruise-control systems
Source: The Economist

Illustration 10: Autonomous car working

Given the risks of potential performance misbehavior it is very important to put some kind of prevention in order to detect the possible executions of the malware In this project we propose using performance signatures to distinguish between intrinsic (innocuous) and extrinsic (malicious) performance variations.

By means of some counters that monitors some CPU variables the system can monitor if the values obtained match those of the performance signatures and if any value exceeds the "correct operation" stop the execution of the program.

2.5 Impact of cache variability In Execution time

Program accesses to memory directly affect the execution time of a program because the time we need to access to cache to get some data is shorter than searching and getting the data directly from memory. So as many Hits in cache we have faster we will execute a program. That is why, against more conflicts, we will need more time to run a program.

CPUs usually include several levels of cache with the intention of minimize the impact of accessing main memory. First level caches are usually small to have reduced access time. However, to allocate more data additional cache space is required. This is why more cache levels are introduced. In this work, the CPU we use used two cache Levels L1 and L2. The second level cache is bigger than L1 but has a longer access time. Still, the time required to access the L2 is much shorter than the one required to access main memory.

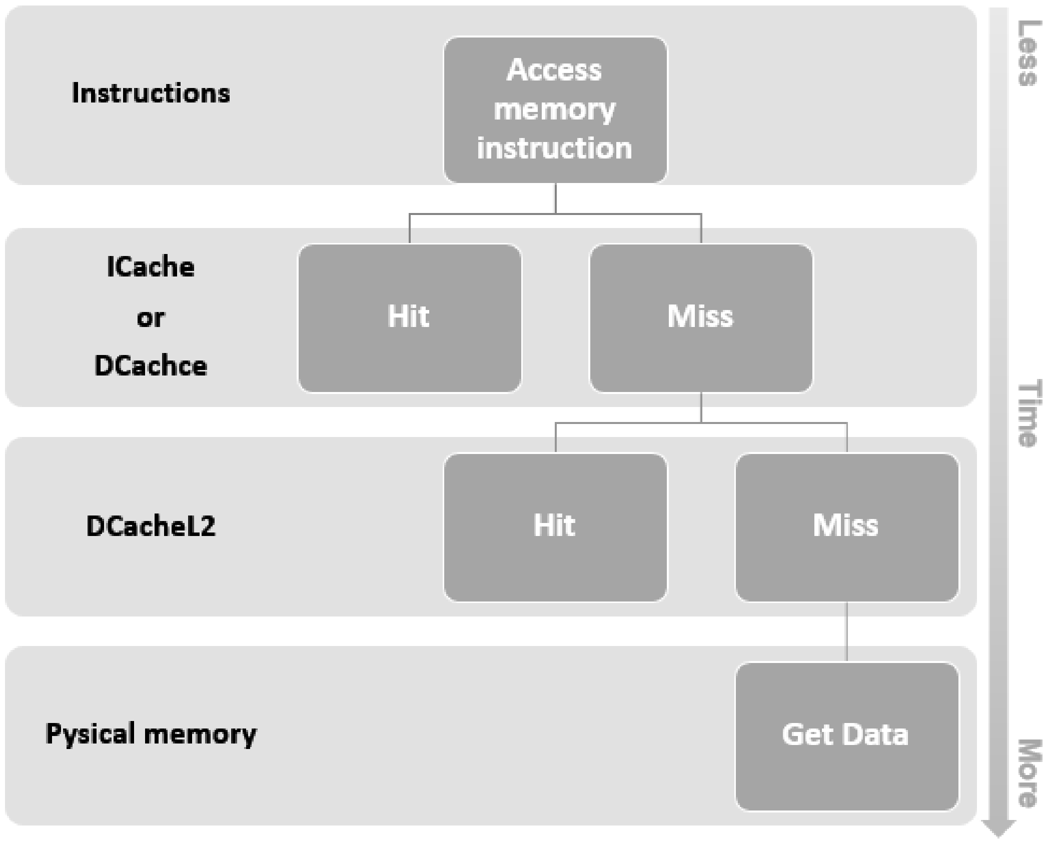


Illustration 11: Schema of cache accessing

3 Performance signatures for security attack detection

Real-time computing (RTC) describes hardware and software systems subject to a "real-time constraint", for example from event to system response. Real-time programs must guarantee response within specified time constraints, often referred to as "deadlines. Systems used for many mission critical applications must be real-time, such as for control of fly-by-wire aircraft, or anti-lock brakes on a vehicle.

Critical applications in the real-time domain have generally a predictable behavior to be able to meet the strict timing deadlines. The determination of whether applications meet timing bounds or not can be studied by monitoring the behavior of the applications when running in the CPU and extracting some statistics and analyzing them. From this analysis phase we can derive performance signatures that we will have to guarantee during the execution. This can be done using performance monitoring counters in the CPU.

The idea is that the performance monitoring unit allows to trigger interruptions based on when a given quota has been exceeded, so we will treat the performance signature as a mechanism to detect anomalous program behavior indicative of a hostile attack, and as a metric to potentially guide the automated development of a patch to correct the program flaws exploited in the attack.

The process of performance monitoring and detecting anomalous behavior is analogous. Data commonly collected for performance monitoring purposes includes the number of references to particular memory locations, the time spent executing program, the frequency and quantity of communication among nodes of a multiprocessor system, and different resources usage. Performance analysis extracts observations of interest to programmers, and displays the conclusions in ways that clearly expose the performance impact of those observations.

In this project we will try to create the performance signatures from these variables:

- Execution time
- Cycles executed
- Executed instructions
- Cache accesses (different levels of cache ICache, DCache & DCacheL2)
- Cache misses (different levels of cache ICache, DCache & DCacheL2)
- Cache hits (different levels of cache ICache, DCache & DCacheL2)

Detecting and correcting anomalous program behavior triggered by an attack can be viewed as a similar process. While excessive or inefficient resource usage raises the red flag in performance tuning, it is deviation from "normal" patterns of operation which signals potential system misbehavior.

Monitoring for anomalous behavior therefore requires defining the variables that might indicate an attack and continually observing these variables during system operation. The values of these variables during program execution constitute the performance signature of the program.

Analyzing this data entails issuing a warning when one or more monitored variables enter a "suspicious" range and aggregating anomaly reports from multiple systems.

In the context of real-time systems we need to know if we are executing malicious code before the execution ends. Thus, we cannot use the execution time, cycles executed and executed instructions to create performance signatures, although we can use all the Cache statistics information.

A performance signature is the set of performance counters that determine the behavior of my application. Given that we focus on single cores. We select cache metrics as the most important metrics to include in the signature. As said before, we exclude execution time since we will need to wait until the end of the “timing budget” or the complete execution to detect the malicious behavior.

In illustration 13 we can see the process we can follow to create performance signatures is composed by 3 steps that are the monitoring of CPU from an execution to obtain some statistics, then the analyzing of this statistics and finally the generation of the CPU performance signatures that can be a threshold of some type of statistics.

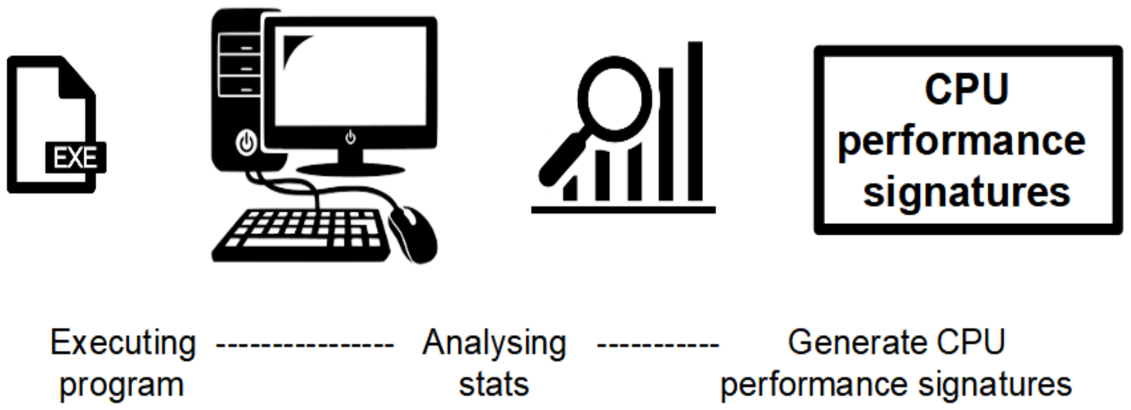

EXE
CPU performance signatures
Executing program
Analysing stats
Generate CPU performance signatures

Illustration 12: CPU performance signature creation process

The process of using CPU performance signatures to security attack detections is composed by the next steps:

- Execute a program in a computer (we assume we have the CPU performance signatures for the program)
- Monitors the CPU during the execution of the program through CPU counters
- Compare the counter with the CPU performance signatures. We can assume that the CPU performance signatures are a threshold that cannot be overcome by the counter values.
- If counter value is less than the threshold we can continue executing the program.
- If counter value overcome the threshold we have to launch a program interruption to stop the execution due to probably we are executing malware.

In Illustration 14 we will be able to see an example of how to use the CPU performance signatures for ICache misses to detect malware executions.

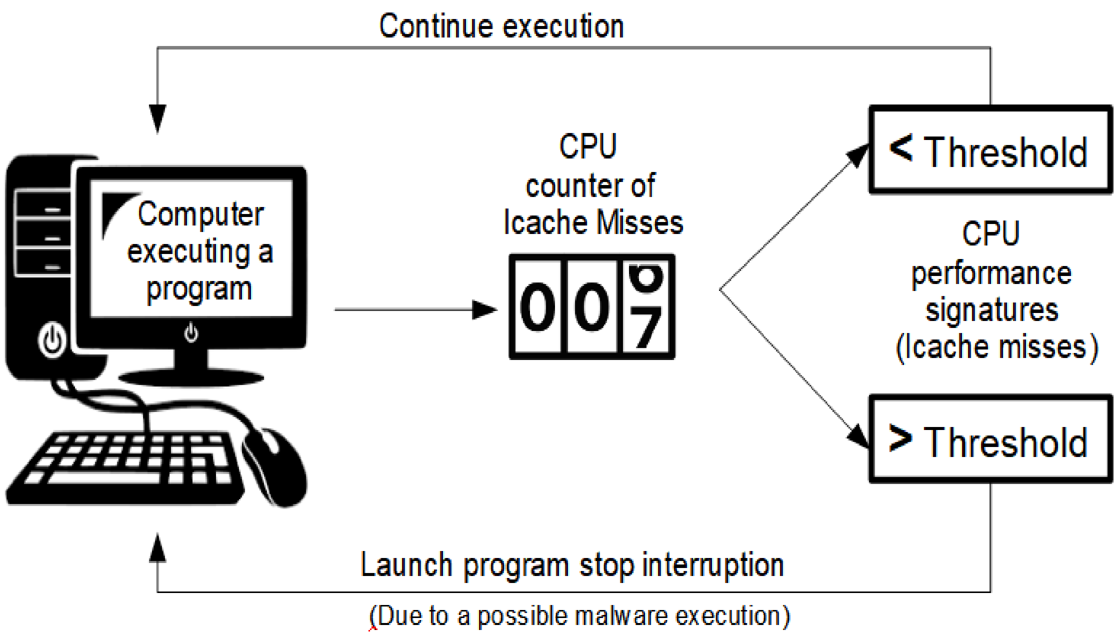

Continue execution
Computer executing a program
CPU counter of Icache Misses
007
< Threshold
CPU performance signatures (Icache misses)
> Threshold
Launch program stop interruption
(Due to a possible malware execution)

Illustration 13: Process of Security attack detection through CPU performance signature

To summarize we proposed the following methodology to implement the detection of security vulnerabilities using performance signatures:

1. We have an analysis phase in which the applications are characterized and the most representative performance metrics are identified. A signature is a set of performance metrics like ICache miss, DCache miss etc.

2. Program the quotas according to the performance signatures so that when the quota of a given performance metric is exceed an interruption is triggered. If the quota is too tight, then we will trigger interruptions (false positives) too frequently if the quota is too high it might be too late to prevent the attack to produce a damage in the system.

3. We have a deployment phase in which the system is working in the real environment. During deployment the monitors check that quotas are never exceeded. Whener a quota is exceeded an interruption is triggered.

4 Methodology

In this section we describe the methodology we have employed to derive performance signatures.

4.1 Resources

In this section we explain carefully all the resources used to derive performance signatures for the EEMB Autobench benchmarks.

4.1.1 Simulator Platform

In order to be able to study the behavior of the CPU performance in different cases, we will use a SPARC simulator resembling the NGMP (Next Generation Microprocessor) processor.

The SPARC simulator simulates with a high accuracy the 4-core NGMP processor, expected to be the target multicore platform for the next European Space Agency missions.

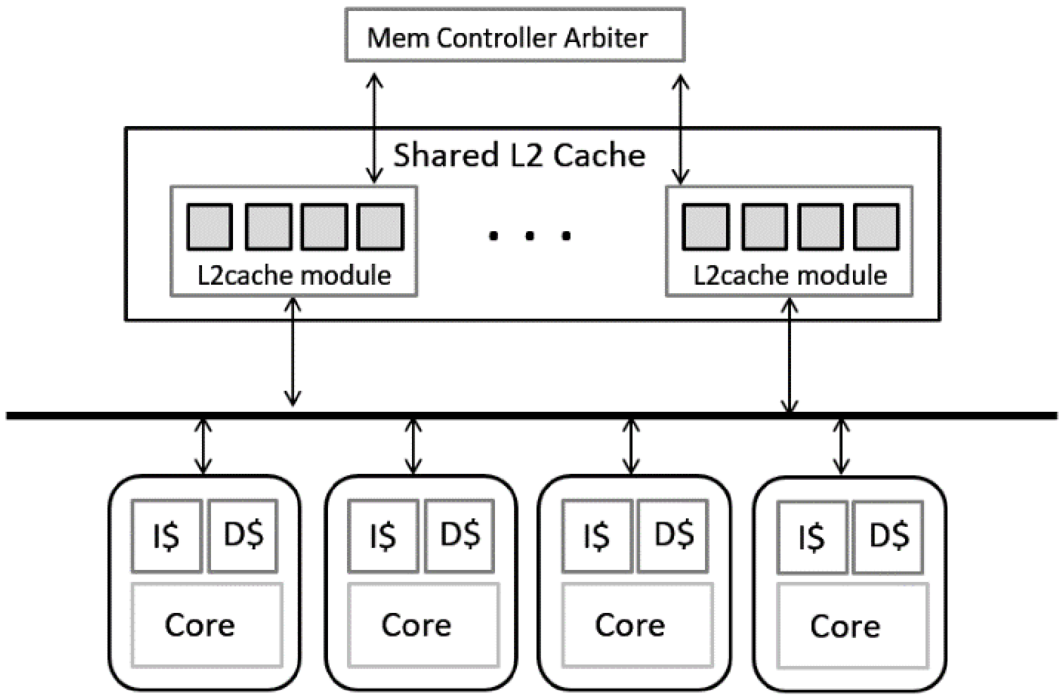


Illustration 14: SPARC architecture scheme

This platform allows us to fill in a sheet of parameters with many possibilities that we have used to execute the benchmarks with different placement and replacement policies of the 3 different caches we have in this the SPARC Simulator (ICache, DCache, UCacheL2).

Illustration 15: Sun Ultra Sparc II

We will use two configurations for the SPARC simulator defined in the parameter file, the COTS (commercial off-the-shelf) and the Time-randomized version [1]

In this project due to the short time we have to perform the study, we will just run executions with 1 core and one benchmark for execution.

	COTS	Randomized
Core	32 bit Sparc ISA 7-stage pipeline FPU	32 bit Sparc ISA 7-stage pipeline Fixed latency FPU
Caches	L1 private	
	4-way 16KB Instructions 4-way 16KB Data LRU replacement Modulo placement	4-way 16KB Instructions 4-way 16KB Data Random replacement RM hRP placement
	L2 shared	
	Unified 128K 4-way LRU Modulo placement	Unified 128K 4-way Random replacement RM hRP placement

Table 1: SPARC COTS & Randomized features

The process to execute a benchmark with this simulator consist of:

```
./sim.exe  param_file  >  output_file
```

- **Sim.exe:** is the executable of the SPARC simulator.
- **Param_file:** is the parameter file where we define the total of CPU (in this study will be always 1), the binary we want to execute and the architecture features of the SPARC.
- **Output_file:** is the file where we want to save all the statistics.

4.1.2 SoCLib

SoCLib is an open platform for virtual prototyping of multi-processors system on chip. This is an Ubuntu OS system given by the project directors that contains all the tools we need to generate the benchmarks, and execute them, including TASA and the SPARC Simulator.

4.1.3 EEMBC Autobench Benchmarks

AutoBench is a suite of 16 benchmarks from EEMBC (embedded microprocessors benchmarks) that allow users to predict the performance of microprocessors and microcontrollers in automotive, industrial, and general-purpose applications. This are the programs that we have used to generate all the statistics for the study. Especially this are the ten benchmarks we are going to use are:

- **Angle to Time Conversion (a2time):** This EEMBC benchmark simulates an embedded automotive application where the CPU reads a counter which measures the real-time delay between pulses sensed from a toothed wheel (gear) on the crankshaft of an engine.
- **Finite Impulse Response (FIR) Filter (aifirf):** This EEMBC benchmark algorithm simulates an embedded automotive/industrial application where the CPU performs a Finite Impulse Response (FIR) filtering sample on 16-bit or 32-bit fixed-point values. Highland low-pass FIR filters simply process the input signal data.
- **Basic Integer and Floating Point (basefp):** This EEMBC benchmark algorithm measures basic integer and floating point capabilities.
- **Bit Manipulation (bitmnp):** This EEMBC benchmark simulates an embedded automotive/industrial application where large numbers of bits have to be manipulated.

- **Cache "Buster" (cacheb):** This EEMBC benchmark simulates an embedded automotive/industrial application without a cache.

- **CAN Remote Data Request (canrdr):** This EEMBC benchmark simulates an embedded automotive application where a Controller Area Network (CAN) interface node exists for exchanging messages across the system.

- **Pointer Chasing (pntrch):** This EEMBC benchmark simulates an embedded automotive/industrial application which performs a lot of pointer manipulation.

- **Pulse Width Modulation (puwmod):** This EEMBC benchmark simulates an application in which an actuator is driven by a PWM signal proportional to some input.

- **Table Lookup and Interpolation (tblook):** This EEMBC benchmark algorithm is used in engine controllers, anti-lock brake systems, and other applications to access constant data quicker than by raw calculation.

- **Tooth to Spark (ttsprk):** This EEMBC benchmark simulates an automotive application where the CPU controls fuel injection and ignition in the engine combustion process.

Figure 16 illustrates the typical execution flow of an EEMBC benchmarks. These benchmarks have an initial phased in which they collect the required inputs, an iterative process in which the actual computation is carried out, and a final phased in which the outputs are produced.

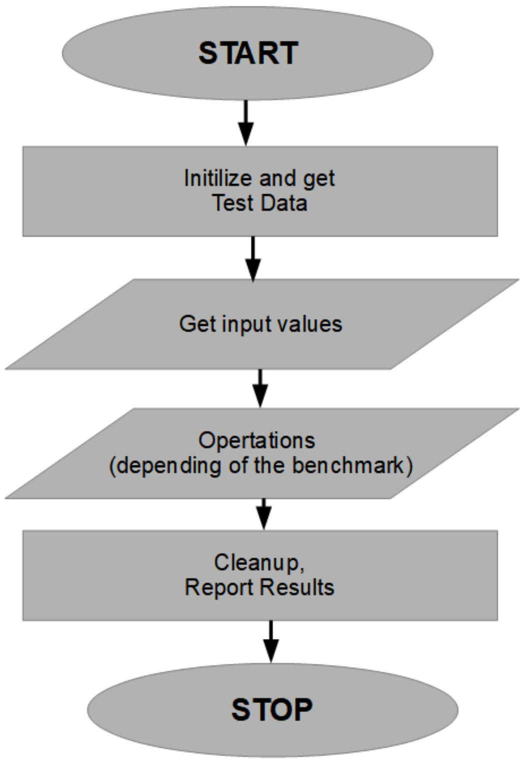
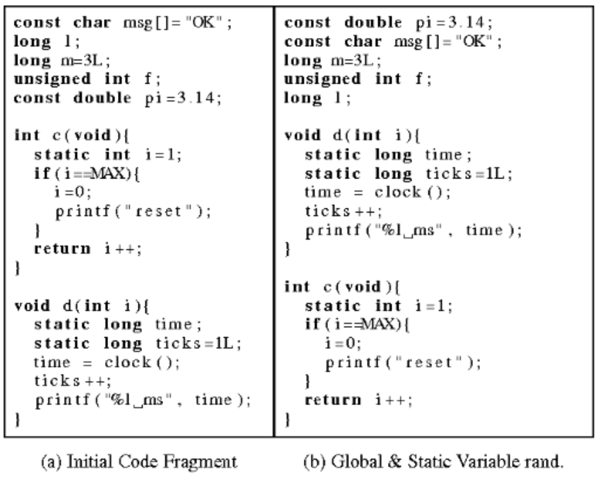


Illustration 16: Schema of all benchmark algorithm flowchart

4.1.4 TASA

TASA (Toolchain-Agnostic Static Software Randomization for Critical Real -Time System) is a compiler given by the project directors that produces binaries of programs with randomized memory layout. That is, given a program (in C) it creates a new identical program, but randomizing the order of declaration of variables, functions, or even creating empty spaces in the stack with the objective of creating a different (random) memory layout in the generated binary. This directly affects the memory layout at program execution, after the binary is loaded in memory. In this way we will be able to run the same programs, but with different cache conflicts between executions.



```
const char msg[]="OK";
long l;
long m=3L;
unsigned int f;
const double pi=3.14;

int c(void){
  static int i=1;
  if(i==MAX){
    i=0;
    printf("reset");
  }
  return i++;
}

void d(int i){
  static long time;
  static long ticks=1L;
  time = clock();
  ticks++;
  printf("%l␣ms", time);
}
```

(a) Initial Code Fragment

```
const double pi=3.14;
const char msg[]="OK";
long m=3L;
unsigned int f;
long l;

void d(int i){
  static long time;
  static long ticks=1L;
  time = clock();
  ticks++;
  printf("%l␣ms", time);
}

int c(void){
  static int i=1;
  if(i==MAX){
    i=0;
    printf("reset");
  }
  return i++;
}
```

(b) Global & Static Variable rand.

Illustration 17: Example of one type of TASA randomization

The process to execute a benchmark with this simulator consist of:

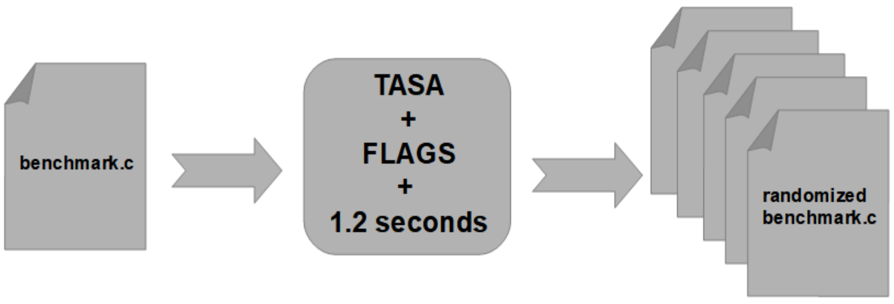


Illustration 18: Scheme of TASA working

./tasa –f –v –g benchmark.c >> randomized_benchmark.c

- **Tasa:** is the executable of Tasa (it exists for 32 or 64 bits).
- **–f –v –g:** this are the flags that we are using to generate the new program.
- **Benchmark.c:** is the original benchmark from which we want to generate more.
- **Randomized_benchmark.c:** is the randomized benchmark generated after executing Tasa.

4.1.5 ARVEI Cluster

This is a high-performance cluster from the UPC DAC (Architecture Computer Department). This cluster has as its function research tasks such as:

- Simulations
- Parallel works
- Intensive calculation
- Executions of a large number of virtual machines

Illustration 19: ARVEI cluster from DAC UPC

In this project we will use this cluster to perform parallel works that will be simulations of EEMBC Autobench benchmarks in a SPARC CPU.

The maximum number of executions that we have performed in parallel have been around 150 because if we increased the number of jobs in parallel, the priority dropped and we did not acquire CPU time optimally.

The user servers have access to the disc of the NAS (the disc scratch centralized of the clusters) and have the same system, which could be used for interactive works in these machines.

The hardware of this cluster is constantly updated and it is composed by different nodes depending of the year in which they were added.

In table 2 we will be able to see the characteristics of the nodes of each year.

Year	Type of nodes	RAM	Memory	Network cards
2006	73 USP 2x Xeon Dual-Core 5148	12 GB	320 GB SATA-2	2 x Intel Pro/1000 Gigabit Ethernet
2010	40 USP 2x Xeon L5630 Dual	24 GB	640 GB SATA-2	2 x Intel Gigabit Ethernet
2012	40 nodes USP 2x Xeon E5-2630L a 2 GHz	64 GB	1 TB SATA-3	4 x Intel Gigabit Ethernet
2014	40 nodes 2x Intel Xeon E5-2630L v2 a 2.40 GHz	128 GB	1 TB SATA-3	4 x Intel Gigabit Ethernet
2016	5 nodes 2x Intel Xeon E5-2630L v3 a 1.80 GHz	128 GB	1 TB SATA-3	2 x Intel Gigabit Ethernet
2017	3 nodes 2x Intel Xeon E5-2630L v4 a 2.20GHz	128 GB	12 TB SATA	2x Intel Gigabit Ethernet
Total	201	484 GB	16 TB	14

Table 2: ARVEI hardware characteristics

The ARVEI cluster is composed of several types of nodes according to the function they have.

- **Access nodes:** these nodes have been used to access to the cluster via SSH and for programing and executing scripts.
  - They allow SSH access to the cluster.
  - From these nodes you can queue jobs.
  - They have access to the centralized user disk and the cluster shared scratch (NAS).
  - They allow to make small developments and smaller tests, but they are not designed for heavy processes.
- **Queue execution nodes:** these nodes have been used to perform the benchmark simulations on SPARC CPU.
  - It can only be accessed remotely through the input nodes.
  - They are part of a private network with access to the outside via Network Address Translation (NAT).
  - They start up and turn off automatically based on their use.
- **Restricted access nodes:** they have not been used
  - The use of these nodes is restricted to a certain group of authorized users.
  - Access is made through a dedicated and restricted queue, which has no limits.
- **Interactive nodes:** they have not been used
  - They have no limitations.
  - The processes that do not fit in the other nodes can be executed.
  - They have the interactive.q queue to access it.

This cluster uses several queues according to the time required by the work we want to execute. In table 3 we can see these queues with their characteristics and their process time limits.

Queue name	Characteristics		Process limit	
	Slot/node	Year nodes	CPU time	Real time
all.q	2	---	3 hours	4 hours
medium.q	1	2006	8 hours	12 hours
	2	2010, 2012, 2014, 2016 & 2017		
big.q	1	2006	48 hours	60 hours
	6	2010		
	10	2012, 2014, 2016 & 2017		
huge.q	1	2006	15 days	16 days
	8	2010		
	12	2012, 2014, 2016 & 2017		
interactive.q	20	arvei-69 to arvei-73	No limits	

Table 3: Schema of ARVEI queues

To perform the EEMBC Autobench benchmarks executions in SPARC simulator, we have used the all.q queue since no execution lasts more than 3 hours and it is the queue with more nodes so we will be able to execute more parallel works and we will have more priority to use a node.

4.1.6 BSC CAOS Remote Machine

This is a remote machine from the Computer Architecture Operating Systems Department (CAOS) from Barcelona Supercomputing Center (BSC). A remote machine user was ceded by the project directors.

These are the characteristics of this remote machine:

- 8 CPU Quad-Core AMD Opteron(tm) Processor 2376 (512 KB as cache size)
- 14 GB of RAM memory
- 62 GB SATA-2 of physical memory
- Intel Corporation 82574L Gigabit Network Connection as network card
- Directly connected with ARVEI cluster

Since this remote machine is shared with other users, the maximum number of CPUs that have been used at the same time are 6, for generating and compiling the benchmarks (explained in 4.2 section) thanks to its direct connection with ARVEI cluster.

4.2 Generation and execution of benchmarks

In this section we describe how we have generated the randomized benchmarks from the original EEMBC Autobench benchmarks, how we have compiled them and finally the process that we have followed to execute them.

4.2.1 Generation and compilation of randomized benchmarks

From the original EEMBC benchmarks provided by the project managers, we have generated N randomized benchmarks have been generated using TASA.

a2time	aifirf	basefp	bitmnp	cacheb	canrdr	pntrch	puwmod	tblook	ttsprk
21500	3000	4000	4000	4000	4000	4000	4000	4800	14500

Table 4: Number of benchmarks generated with TASA

Different number of randomized benchmarks have been created depending on the benchmark in such a way that we can get a minimum of 1000 correct executions from each benchmark.

The correct executions are those which have the same number or very similar of instructions executed than the original benchmark. On the other hand, we have the wrong executions where the number of executed instructions is less than the original benchmark. These wrong executions happen because of the randomization of TASA that does not take into account the fact that you have to initialize a variable before calling it, and some other similar errors.

After all the randomized benchmarks have been generated, they had to be compiled to be able to execute in the SPARC simulator.

To minimize the time of this process two C++ script has been created, one for the generation the randomized benchmarks with TASA and the other for generating the binaries for a SPARC architecture.

The generation script it takes 1.2 seconds for every benchmark generated, although it allows to generate all the ten benchmarks in parallel.

The compiling script it takes 1 second for every three benchmarks, but it not allows to generate all the ten benchmarks in parallel.

During this process it had to face two problems:

- The time needed to compile all the benchmarks
- The upload time of all the binaries to the cluster used to execute them

The founded solution consisted on using a remote BSC machine to perform this process, because this remote machine had 6 useful CPU and I had been able to compile all the benchmarks in parallel in such a way to divide by 6 the compiling time. Furthermore, this remote machine is directly connected with the cluster that was used to execute the benchmarks, in such a way that to pass all the binaries from the remote machine to the cluster is much faster than from the personal computer to the cluster via internet.

In table 5 we will be able to see the time (seconds) needed to generate and compile all the benchmarks for a version, in this case is for the normal version.

Process	Generation (parallel)	Compilation (no parallel)	Total
a2time	25800	7167	32967
aifirf	3600	1000	4600
basefp	4800	1333	6133
bitmnp	4800	1333	6133
cacheb	4800	1333	6133
canrdr	4800	1333	6133
pntrch	4800	1333	6133
puwmod	4800	1333	6133
tblook	5760	1600	7360
ttsprk	17400	4833	22233
Total	25800	22598	48398

Table 5: Generation and compilation time (in seconds) of the benchmarks

4.2.2 Execution of the benchmarks

Once all the binaries had been generated, they had to be executed with the SPARC simulator. The necessary resources to carry out executions with the simulator are:

- **SPARC simulator executable:** provided by the project directors within the OS SoCLib.
- **Benchmark binary:** generated in the process previously explained.
- **Parameter file:** this is a file that contains all the information needed to execute something in the simulator in addition to many architecture features. Two versions of parameter file had been created changing the placement and replacement polices.

In table 6 we will be able to see the placement and replacement policies (explained in 2.3 section) that we are using to execute the benchmarks with COTS and Time randomized processor implementation.

Type		COTS	Time randomized processor
Replacement policy	ICache	LRU	random eviction on miss
	DCache	LRU	random eviction on miss
	ICacheL2	LRU	random eviction on miss
	DCacheL2	LRU	random eviction on miss
Placement policy	ICache	modulo	random on run
	DCache	modulo	enhanced random modulo
	ICacheL2	modulo	random on run
	DCacheL2	modulo	random on run

Table 6: Placement and replacement policies of SPARC

Caches

The biggest problem that it has to face in this process has been the long time needed to be able to execute all the benchmarks. The solution applied was performing this process in ARVEI cluster that had lot of CPUs and has allowed us to perform about 150 parallel benchmarks executions. A python script has been created to automate the process of executing the benchmarks.

In table 7 we can see the execution times (mean in seconds and totals in hours) that we have needed to execute all the benchmarks.

Type of execution	COTS		Time randomized processor		TOTAL
	Mean	Total	Mean	Total	
a2time	1081,920	300,533	1137,974	316,104	616,637
aifirf	1512,517	420,144	1477,423	410,395	830,539
basefp	2539,630	705,453	2441,275	678,132	1383,585
bitmnp	6393,301	1775,917	7598,990	2110,831	3886,748
cacheb	550,936	153,038	625,004	173,612	326,650
canrdr	1001,260	278,128	1198,374	332,882	611,009
pntrch	2887,026	801,952	2387,878	663,299	1465,251
puwmod	969,151	269,209	845,100	234,750	503,959
tblook	1647,692	457,692	1328,828	369,119	826,811
ttsprk	1721,719	478,255	1458,295	405,082	883,337
Total CPU time	20305,152	5640,320	20499,141	5694,206	11334,526

Table 7: Execution times of all benchmarks (in seconds & hours)

In table 8 we can see the mean of the computational time in cycle that we have needed to execute every benchmark the benchmarks, we also can see in the last column the total of cycles needed to execute all the benchmarks.

Type of execution	COTS (mean)	Time randomized processor (mean)	TOTAL
a2time	211452326,4	223533773,6	434986100000
aifirf	233742784,7	234461433,5	468204218200
basefp	469475839,1	472469024,7	941944863800
bitmnp	1244832893	1269665045	2514497938000
cacheb	100102196,6	100502531,5	200604728100
canrdr	186196520	186627837,8	372824357800
pntrch	375111018,7	375171365,8	750282384500
puwmod	132193921,3	133032506	265226427300
tblook	228483853,7	235954945	464438798700
ttsprk	237720206,1	241492129,7	479212335800
Total CPU time	3419311559,6	3472910592,6	6892222152200

Table 8: Computation requirements (in cycles)

Through this table we will be able to know the amount of CPU time that we have required from the high-performance ARVEI cluster.

4.3 Treat of the stats

In this section it will be explained the process of treating the statistics from the output of the simulator to the generation of the different Excel plots and tables.

4.3.1 Output to stats

The output obtained from the simulator after the execution of a benchmark is very long and contains lot of information that we are not going to use for this project. In order to treat and filter in an easier way the statistics we needed the important data to Excel.

The data that we wanted for this project from the outputs was:

- Execution time
- Cycles executed
- Executed Instructions
- ICache:
    - Read Accesses
    - Read Hits
    - Read Misses
- DCache:
    - Write & Read Accesses
    - Write & Read Hits
    - Write & Read Misses
- DCacheL2:
    - Write & Read Accesses
    - Write & Read Hits
    - Write & Read Misses

We don't take into account the ICache Write because the accesses are always 0.

In the first instance, to convert this data to an XLS spreadsheet a C++ script, with some functions from an external library called LibXL, was used. At first it worked fine, but as you worked with a higher amount of data it took too time to generate the spreadsheet.

For that reason, the solution that was applied consist on generating a CSV document, given its ease of generation and the short time it takes.

Finally, we will manually copy and paste the CSV data into an Excel with an own customized template to make it easier to see the data (since there are lot of data around 18000 values for every benchmark).

4.3.2 Filtered of stats

Once we have all the statistics on an Excel document a filtered of them was needed to be able to analyze them in an easier way.

To perform this process some C++ scripts have been created:

- **Filter between good and wrong executions:** this script generates a CSV file with all the stats but just from the executions where the number of instructions executed is very similar to the original benchmark.

- **Generator of IPC (instructions per cycle):** this script generates a CSV file with the IPC of all the executions (we can get the IPC dividing the number of executed instructions by the number of cycles) and it also allows you to filter the executions between good and wrong.

- **Generator of Hit/Miss ratio:** this script generates a CSV file with the Hit/Miss ratio on percentage of ICache reads, DCache reads & writes, DCacheL2 reads & writes of every execution. It also allows you to filter the executions between good and wrong.

- **Generator of Miss/1000 inst.:** this script generates a CSV file with the Misses per 1000 instructions of ICache reads misses, DCache reads & writes misses, DCacheL2 reads & writes misses of every execution. It also allows you to filter the executions between good and wrong.

- **Generator of Minimum, Maximum, Mean, Median, Mode, Typical deviation, Mean Absolut deviation, Variance:** this script generates these variables for all columns from a CSV that we have to pass as a parameter.

4.3.3 Generation of plots

When all the stats were well filtered in an Excel document, it was time to generate all the plots to study the behavior of the CPU depending on the execution. These plots have been made using Excel due to the ease with which you can generate and manage the plots with this program. The plots that were generated are:

- IPC plots

- ICache plots (Misses/kinst.)

- DCache plots (Misses/kinst.)

- DcacheL2 plots (Misses/kinst.)

4.4 Schema of all process

In table 9 we will be able to see a summary of the process we have followed to collect and treat all the necessary statistics to carry out this project.

Benchmarks generation	• Generate normal randomized benchmarks for the two versions with TASA from EEMBC Autobench benchmarks (in BSC remote machine)
Binaries generation	• Compile normal randomized benchmarks for SPARC architecture (in BSC remote machine)
Benchmarks execution	• Execute normal benchmarks for COTS cache implementation in SPARC simulator (in ARVEI cluster) • Execute normal benchmarks for Time randomized processor cache implementation in SPARC simulator (in ARVEI cluster)
Output to Stats	• Generate CSV with the important data from normal benchmarks for COTS cache implementation executions outputs (in ARVEI cluster) • Generate CSV with the important data from normal benchmarks for Time randomized processor cache implementation executions outputs (in ARVEI cluster)
Stats filtering	• Generate Excel file for good executions from CSV of COTS normal benchmarks • Generate Excel file for bad executions from CSV of COTS normal benchmarks • Generate Excel file for good executions from CSV of Time randomized processor normal benchmarks • Generate Excel file for bad executions from CSV of Time randomized processor normal benchmarks
Treat of Stats	• Generation of IPC for all Excel • Generation of Hit/Miss ratio for every cache of every Excel • Generation Miss/kinst ratio for every cache of every Excel • Generate the minimum, maximum, mean, median, mode, typical deviation, mean Absolut deviation, variance, quartile (3), percentile (0,9) of every Excel
Generation of Plots	• Generation of IPC plots from IPC Excel • Generation of ICache plots (Misses/kinst.) from Miss/Kinst Excel • Generation of DCache plots (Misses/kinst.) from Miss/Kinst Excel • Generation of DCacheL2 plots (Misses/kinst.) from Miss/Kinst Excel

Table 9: Schema of collecting stats process

5 Evaluation

In this section it is describe the behavior of SPARC with the EEMBC Autobench benchmarks by means of the extracted statistics.

5.1 Executions of EEMBC in SPARC with COTS implementation

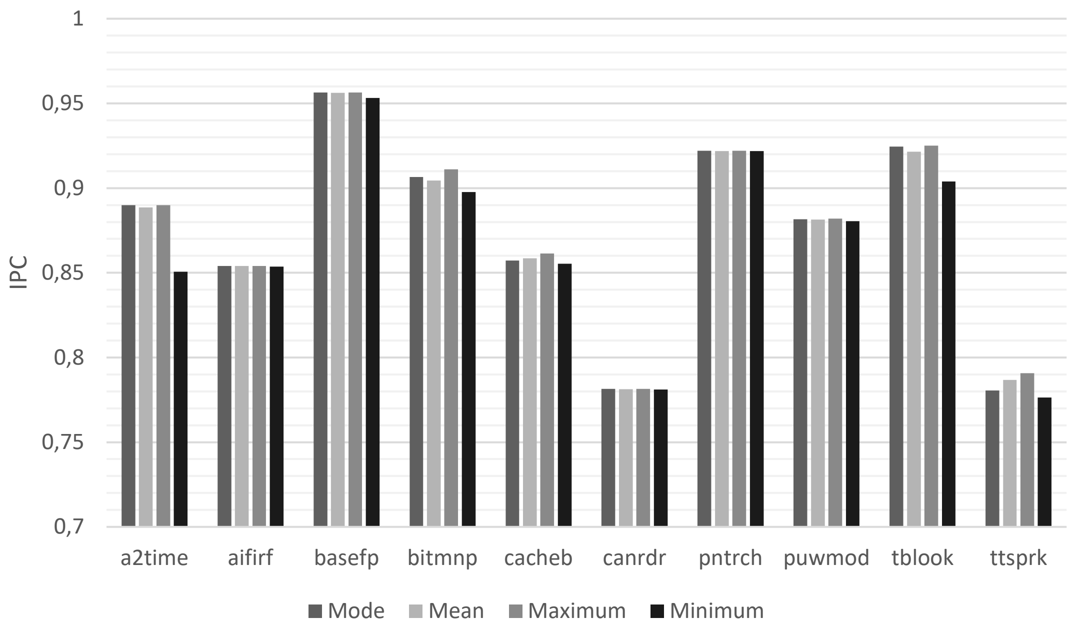
This section explains the statistics obtained by executing the workloads in SPARC simulator with COTS implementation.

5.1.1 Character of EEMBC in COTS

This section describes the character of the SPARC simulator with COTS implementation when executing the different workloads.

5.1.1.1 IPC analysis

In plot 1 we can see the mode, mean, maximum and minimum IPC of 1000 good executions of every benchmark for COTS implementation.

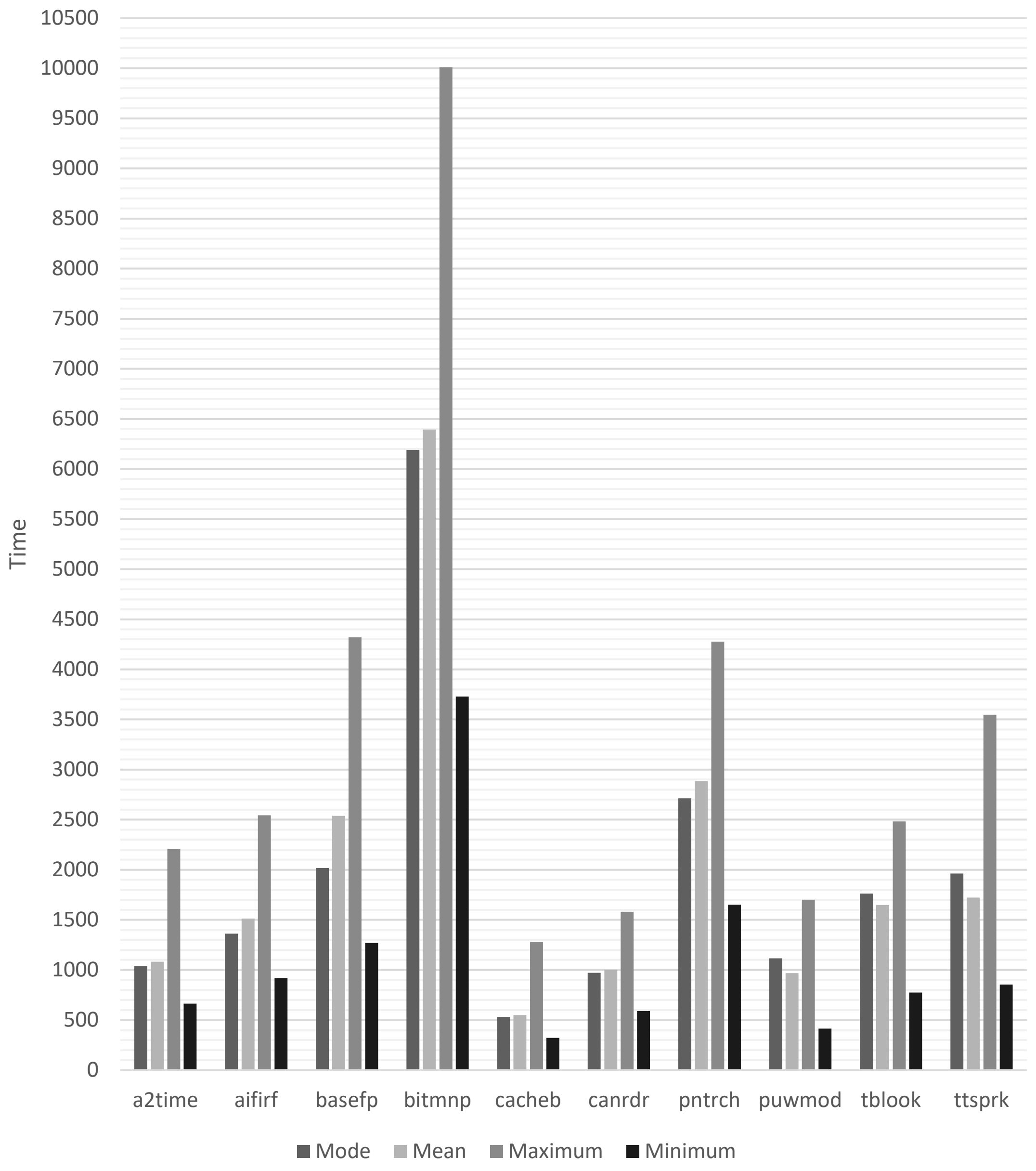


Plot 1: IPC of all benchmarks for COTS (1000 executions)

All the values are really similar for the same benchmark so it shows that the good execution of this benchmarks will have little variabilities for IPC, subtracting some concrete cases as the minimum of *a2time* or *tblook* that are very low and it could be considerate as a pathological cases.

5.1.1.2 Execution time analysis

In plot 2 we can see the mode, mean, maximum and minimum of execution times of 1000 good executions of every benchmark for COTS implementation.



Plot 2: Time in seconds of all benchmarks for COTS (1000 executions)

In one hand, in this plot unlike the previous we can see that the maximums and the minimums have values very different from the mode, because of this we can assume that it will be more pathological cases than in IPC plot 1.

In the other hand, we can see little variabilities between the mean and mode, this will shows that most of the execution times are very similar and that the pathological cases are really far from expected execution times.

5.1.2 ICache Misses

To create the performance signatures, we have focused on the ICache that is the more important cache due to that is the one which is accessed by the instructions.

We have also chosen the ICache because we want to know if we are executing a malicious program before it finishes so we can create performance signatures with the number of misses and when the number of misses overcome the threshold we could stop the execution through throwing an interruption.

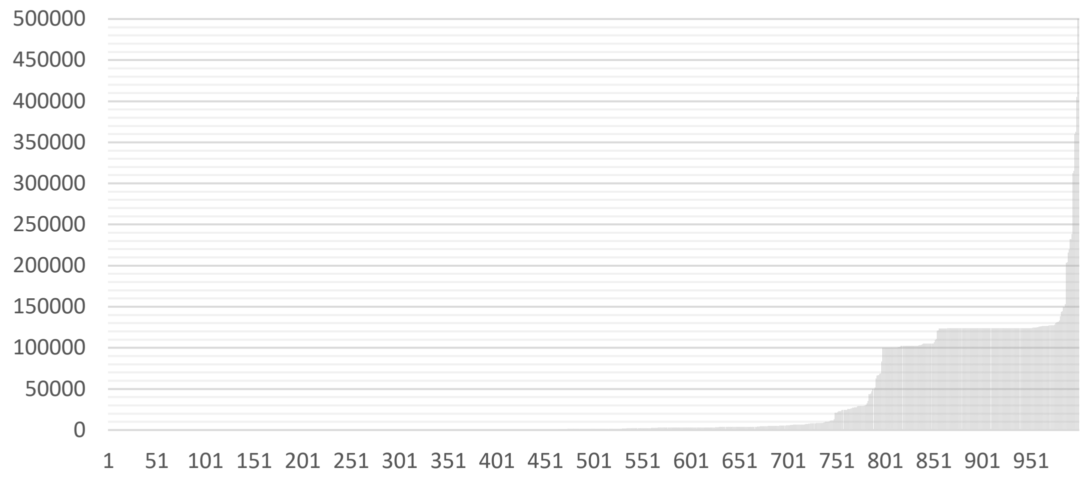
In table 11 it will be able to see the statistics that summarize the set of raw misses for all the executions with a COTS architecture for each benchmark.

Benchmark	mode	median	mean	quartile (3)	percentile (0.9)	maximum
a2time	447	1349	30751,874	21148	123863,2	1040962
aifirf	445	417	394,702	444	447	452
basefp	472	453	5464,595	472	476	160292
bitmnp	2244605	2247332	2246274,717	2264831,75	2280171	2319040
cacheb	481	366,5	337,999	471,25	481	488
canrdr	361	345,5	322,34	361	363	367
pntrch	549	517	450,241	548	551	557
puwmod	438	410,5	388,713	437	439,2	445
tblook	31205	78124	91987,944	123421,25	183725,6	553484
ttsprk	934	95354	114901,049	171633	248510	432323

Table 10: ICache Raw misses for every benchmark for COTS

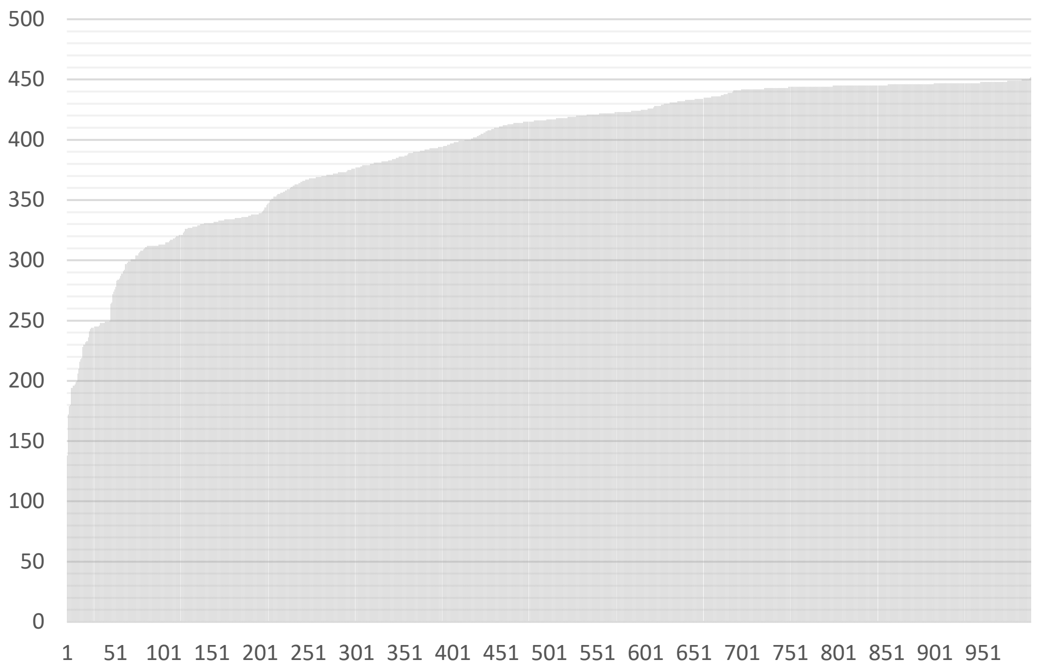
In the next 10 plots we will see the number of ICache Raw misses from all benchmarks, we could see that most of them have very similar shape.

- a2time



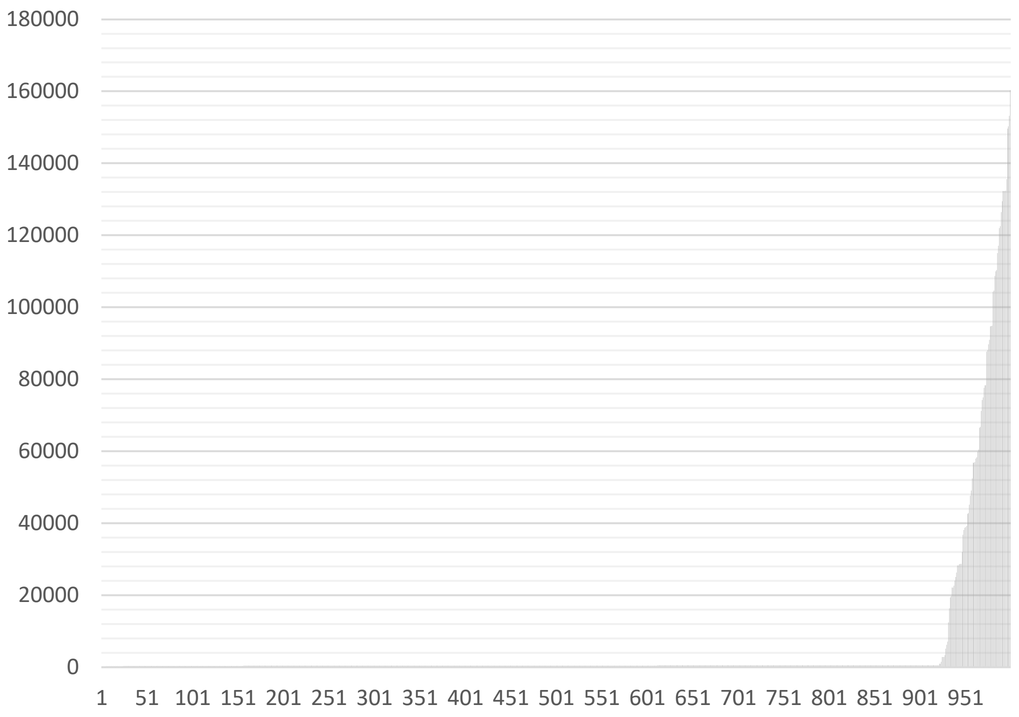
Plot 3: ICache raw misses of a2time for COTS

- aifirf



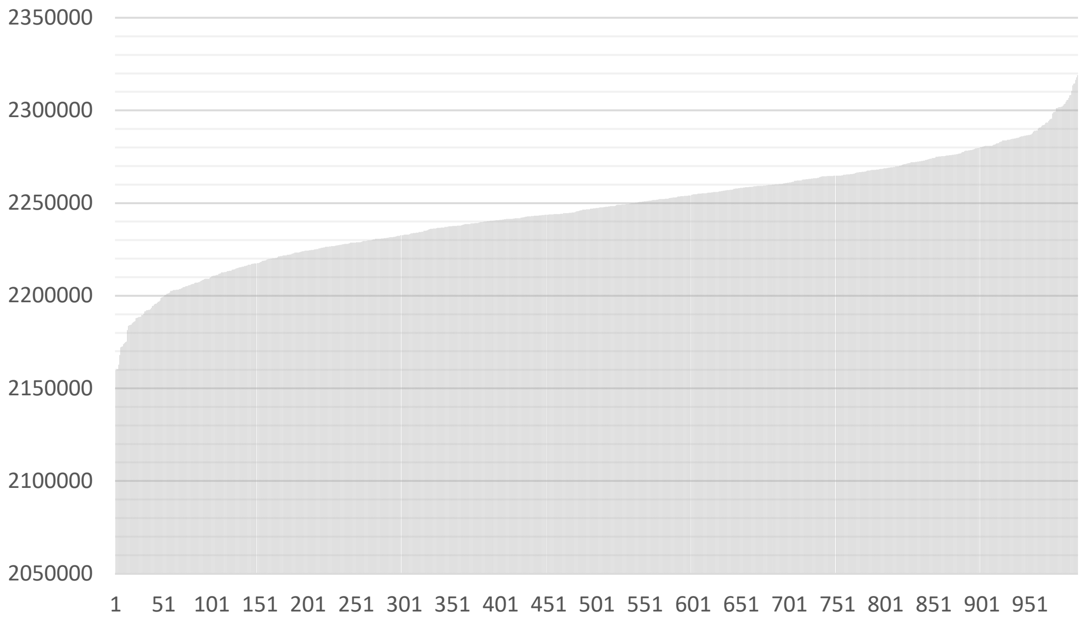
Plot 4: ICache raw misses of aifirf for COTS

- basefp



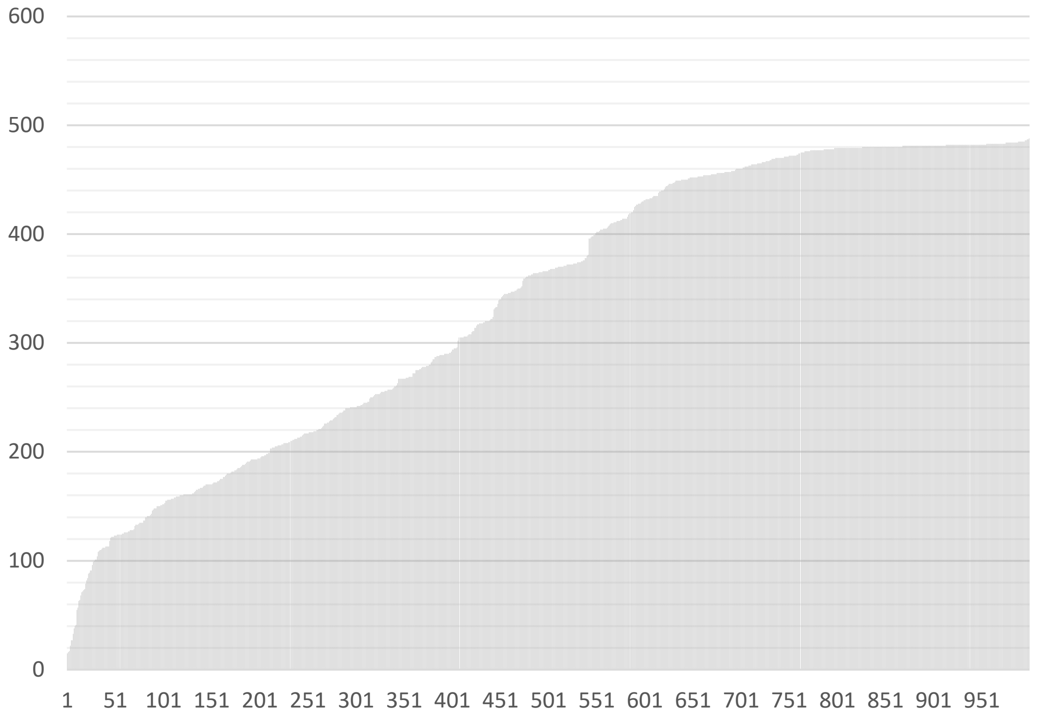
Plot 5: ICache raw misses of basefp for COTS

- bitmnp



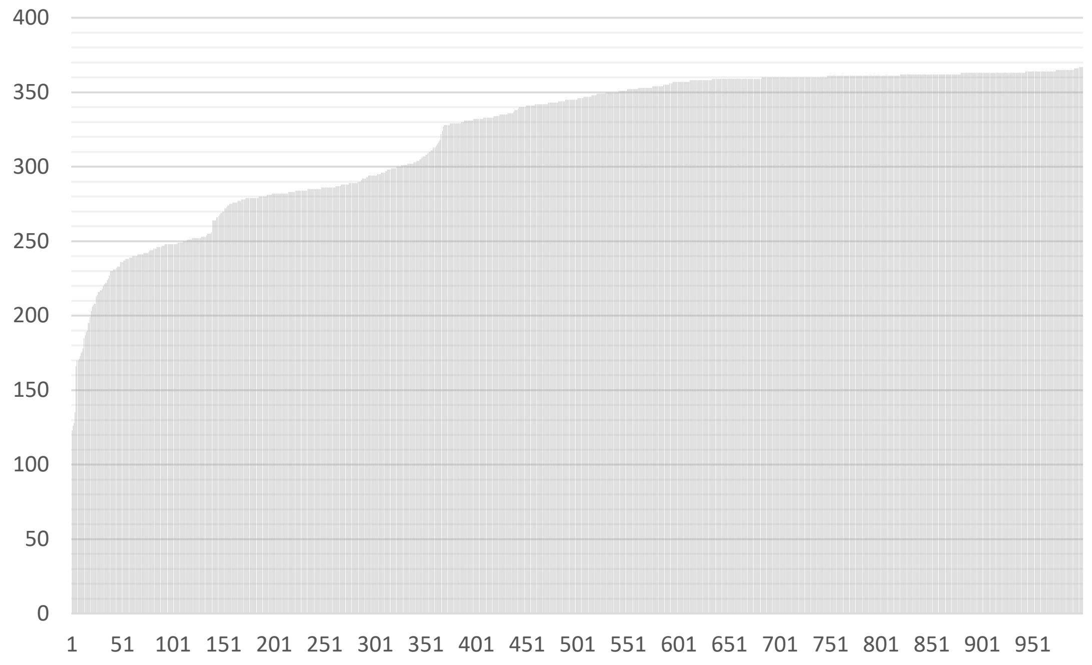
Plot 6: ICache raw misses of bitmnp for COTS

- cacheb



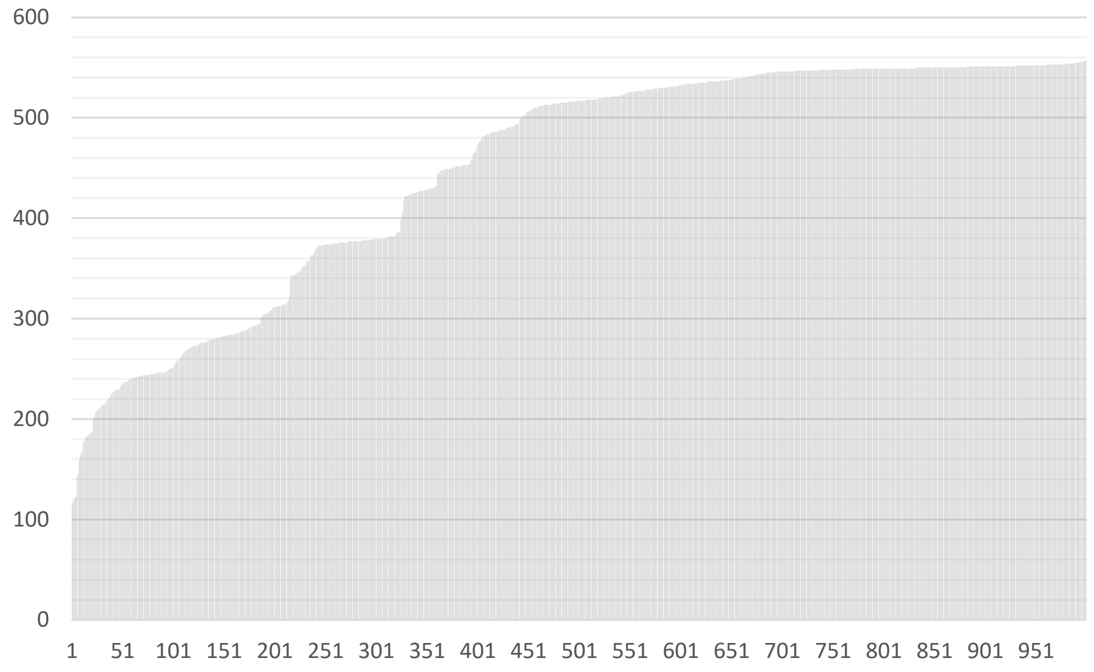
Plot 7: ICache raw misses of cacheb for COTS

- canrdr



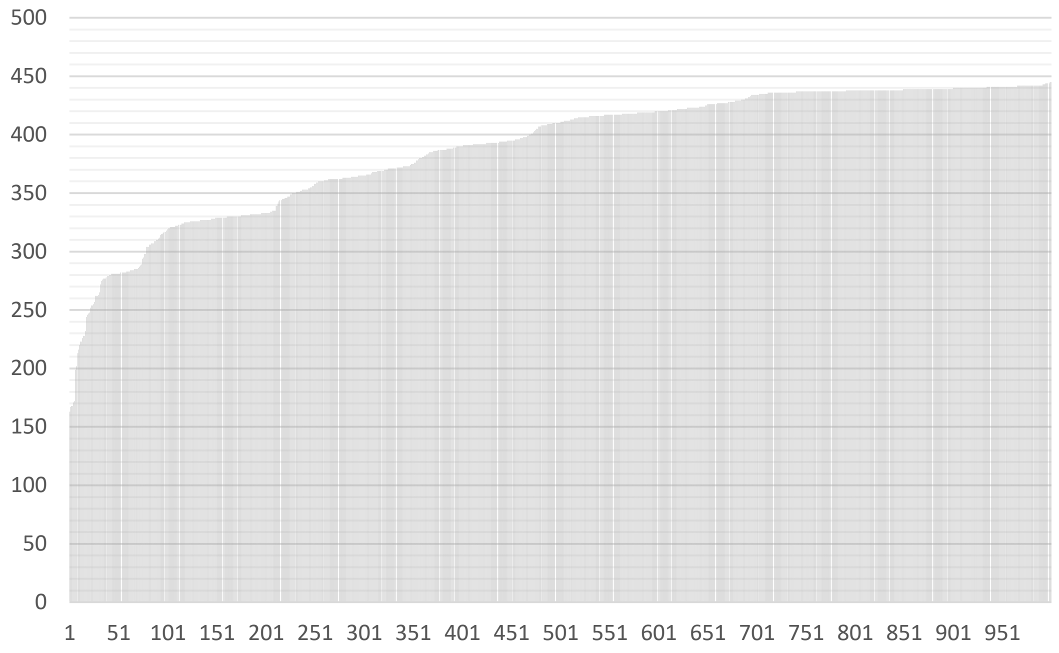
Plot 8: ICache raw misses of canrdr for COTS

- pntrch



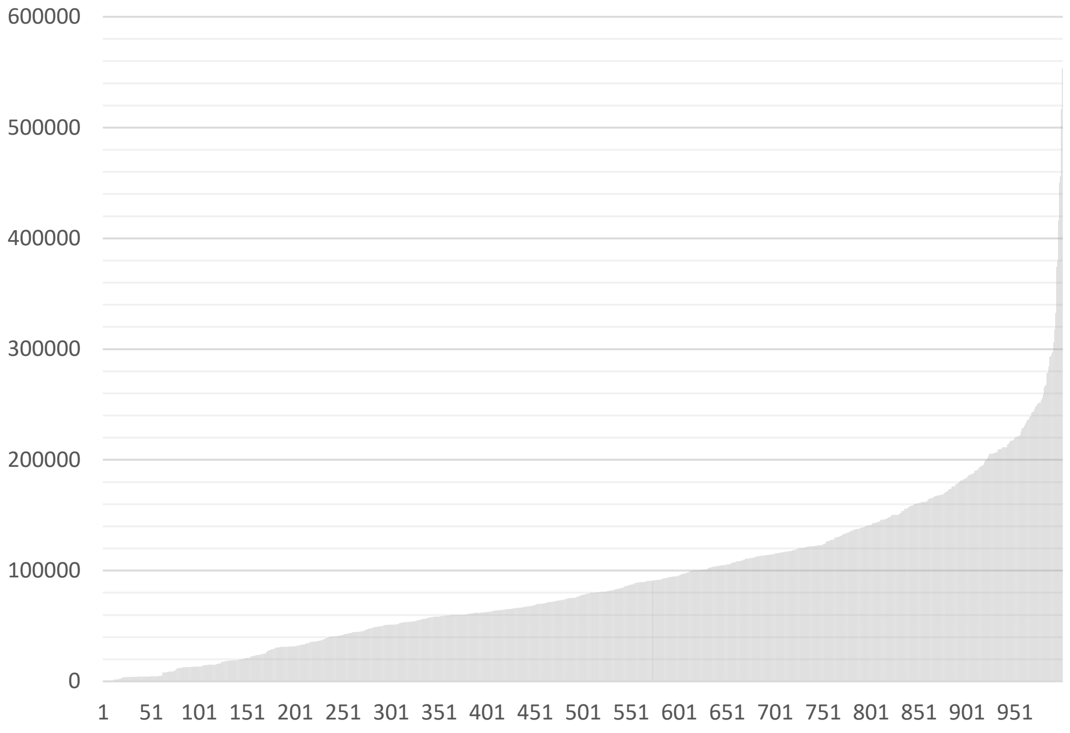
Plot 9: ICache raw misses of pntrch for COTS

- puwmod



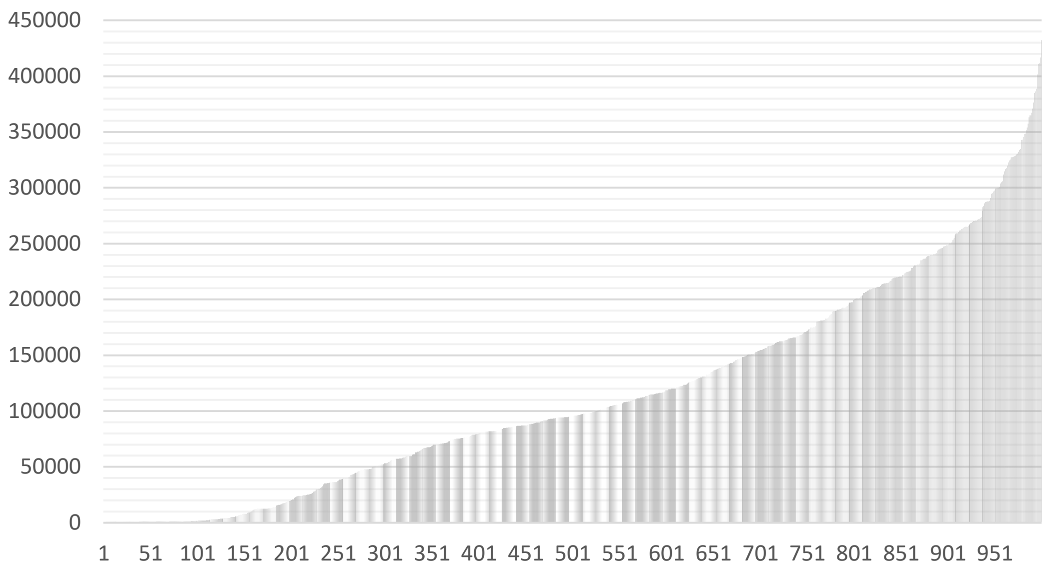
Plot 10: ICache raw misses of puwmod for COTS

- tblook



Plot 11: ICache raw misses of tblook for COTS

- ttsprk



Plot 12: ICache raw misses of ttsprk for COTS

To summarize the last plots, we can tell that six of them have normal COTS shapes and for that reason it will be easy to generate the performance signatures. On the other hand we have the a2time, basefp, tblook and ttsprk that have very different from the normal COTS shapes and that is because we have very separated maximums and minimums, for that reason we will have less precision on our performance signatures.

In table 11 we can see the coefficient of variation (%) for ICache misses for all the benchmarks tested with a COTS architecture. In this table we can see that the bigger Coefficient of variation are from the benchmarks with no normal plot shapes.

Benchmark	Coefficient of variation (%)
a2time	224,0562386
aifirf	14,753806
basefp	391,1250336
bitmnp	1,2128514
cacheb	39,3491999
canrdr	14,8687537
pntrch	26,1894014
puwmod	14,1445448
tblook	76,8111995
ttsprk	81,9354015

Table 11: Coefficients of variation (%) ICache miss for COTS

5.1.3 Evaluating performance signatures

In this section it will be evaluate the performance signatures from the statistics showed in the previous section.

5.1.3.1 Signature definition

After analyzing all the collected stats and as we have explained before, we will create the performance signature from the ICache raw misses.

In table 12 we can see the number of pathological cases that we can find depending on the number of misses for every benchmark with COTS implementation. For example 2X while tell the number of executions that overcome twice or more the mode, and the same for the fifth, tenth and hundredth.

benchmark	2X	5X	10X	100X
a2time	537	466	329	214
aifirf	0	0	0	0
basefp	77	76	72	45
bitmnp	0	0	0	0
cacheb	0	0	0	0
canrdr	0	0	0	0
pntrch	0	0	0	0
puwmod	0	0	0	0
tblook	602	162	9	0
ttsprk	898	864	844	520

Table 12: Pathological cases compared with mode of ICache Raw misses for COTS

In table 13 we can see the same as in table 12, but instead of using mode to filter we will be using the mean.

benchmark	2X	5X	10X	100X
a2time	210	14	7	0
Aifirf	0	0	0	0
Basefp	69	59	42	0
Bitmnp	0	0	0	0
Cacheb	0	0	0	0
Canrdr	0	0	0	0
Pntrch	0	0	0	0
Puwmod	0	0	0	0
Tblook	100	2	0	0
Ttsprk	136	0	0	0

Table 13: Pathological cases compared with mean of ICache Raw misses for COTS

In tables 14 and 15, we can see the average of interruptions triggered for the false positives with mode and mean as a reference.

benchmark	2X	5X	10X	100X
a2time	0,924838102	0,424883604	0,296527891	0,043160446
Aifirf	0	0	0	0
basefp	6,02781726	2,442121638	1,285891492	0,17624516
bitmnp	0	0	0	0
cacheb	0	0	0	0
canrdr	0	0	0	0
Pntrch	0	0	0	0
puwmod	0	0	0	0
Tblook	0,715712203	0,464780249	0,458664207	0
Ttsprk	0,556345703	0,231084874	0,118139344	0,01615867

Table 14: Average of interruptions triggered for false positives with mode as a reference for COTS

benchmark	2X	5X	10X	100X
a2time	150,2284116	168,6053052	119,1912752	0
Aifirf	0	0	0	0
basefp	77,38220032	34,69046969	21,10756659	0
bitmnp	0	0	0	0
cacheb	0	0	0	0
canrdr	0	0	0	0
Pntrch	0	0	0	0
puwmod	0	0	0	0
Tblook	3,873990226	3,428607595	0	0
Ttsprk	154,6655789	0	0	0

Table 16: Average of interruptions triggered for false positives with mean as a reference for COTS

5.2 Executions of EEMBC in SPARC with Time randomized processor implementation

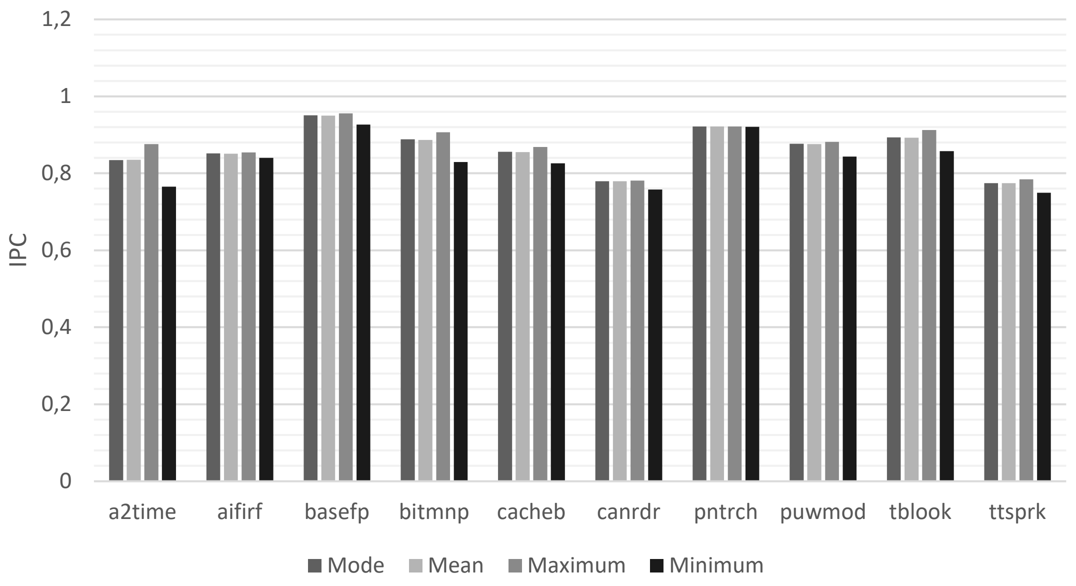
This section explains the statistics obtained by executing the workloads in SPARC simulator with Time randomized processor implementation.

5.2.1 Character of EEMBC in Time randomized processor

This section describes the character of the SPARC simulator with Time randomized processor implementation when executing the different workloads.

5.2.1.1 IPC analysis

In plot 13 we can see the mode, mean, maximum and minimum IPC of 1000 good executions of every benchmark for Time randomized processor implementation.



Plot 13: IPC of all benchmarks for MBPTA (1000 executions)

All the values are really similar for the same benchmark, even so the values are more variant than COTS IPC plot (plot 1) due to for this benchmarks randomized caches are not favorable.

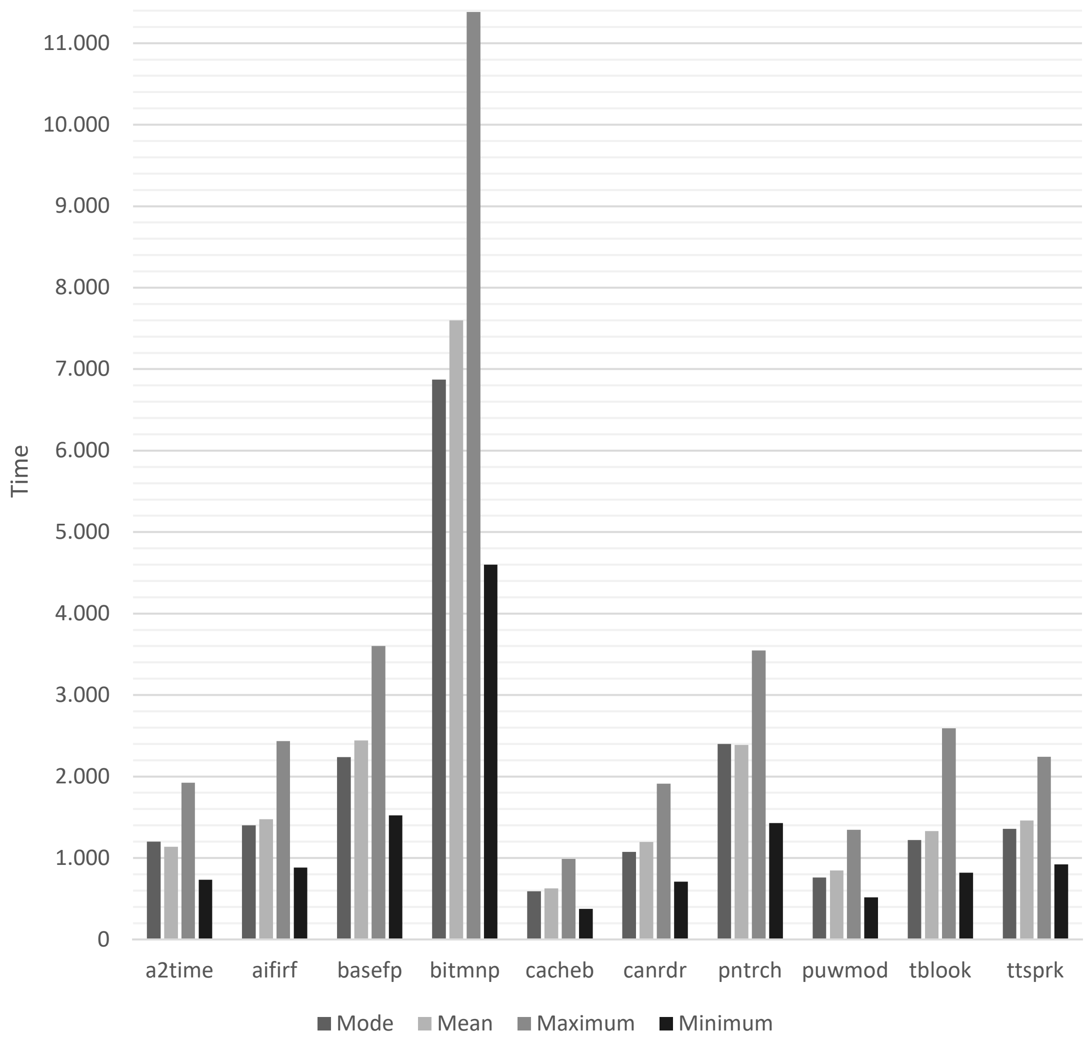
We can see from the plot that good executions of this benchmarks will have little variabilities for IPC.

On one hand, we find some benchmarks we can considerate some pathological cases in concrete cases where the minimum and the maximum are very different from the mode.

On the other hand we have some benchmarks like *pntrch* that don't have any variability between the mode, mean, maximum and minimum.

5.2.1.2 Execution time analysis

In plot 14 we can see the mode, mean, maximum and minimum of execution times of 1000 good executions of every benchmark for Time randomized processor implementation.



Plot 14: Time in seconds of all benchmarks for MBPTA (1000 executions)

In one hand, in this plot unlike the IPC plot we can see that the maximums and the minimums have values very different from the mode, because of this we can assume that it will be more pathological cases than in IPC plot 14.

In the other hand, we can see little variabilities between the mean and mode, this will shows that most of the execution times are very similar and that the pathological cases are really far from expected execution times.

Comparing this plot with the COTS execution time plots we can see that most of the benchmarks have a bigger mean value so it is slower to execute with Time randomized processor than with COTS, but it is less difference between pathological cases (very different maximums and minimums) for Time randomized processor than for COTS.

5.2.2 ICache Misses

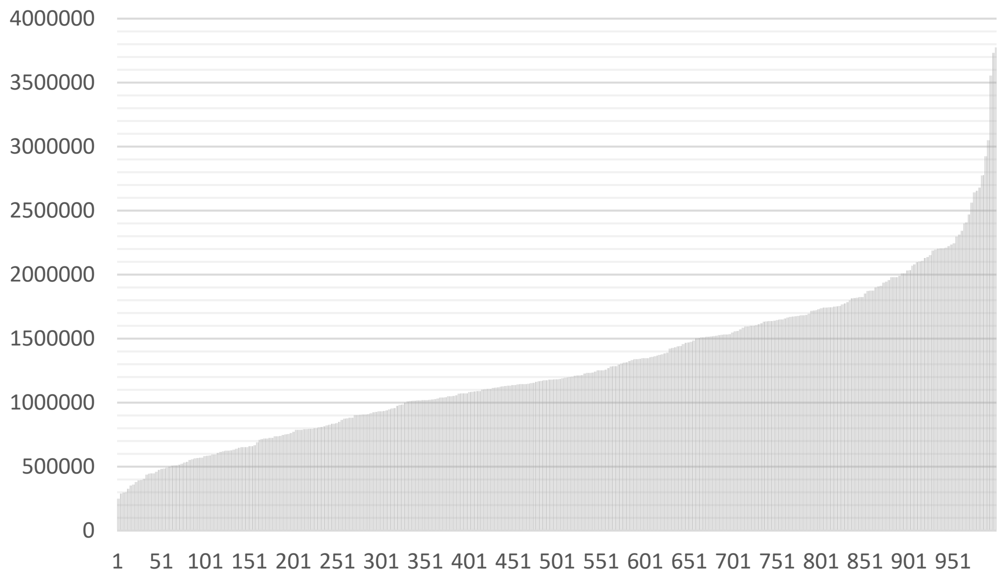
In table 12 it will be able to see the statistics that summarize the set of raw misses for all the executions with a Time randomized processor architecture for each benchmark.

Benchmark	mode	median	mean	quartile (3)	percentile (0.9)	maximum
a2time	1242208	1182610	1283042,023	1645731	2035035	3775651
aifirf	452	61981	75637,187	104951	161788,4	398988
basefp	207356	285205,5	323928,322	399335,5	553234,3	1648835
bitmnp	3833322	4518100,5	4787291,072	5398464,25	6551016	14858991
cacheb	57770	89160,5	99263,669	127816,75	173136,5	425229
canrdr	401	8039	21798,114	24034,25	55104,8	641340
pntrch	639	681,5	5557,538	6876,5	16746,5	55661
puwmod	28132	30820	40744,97	52715,5	80072,8	587044
tblook	842587	856623	880300,077	1011055,5	1155527,6	1826010
ttsprk	508552	513043	529938,263	591282,5	683120,6	1356484

Table 14: ICache Raw misses for every benchmark for Time randomized processor

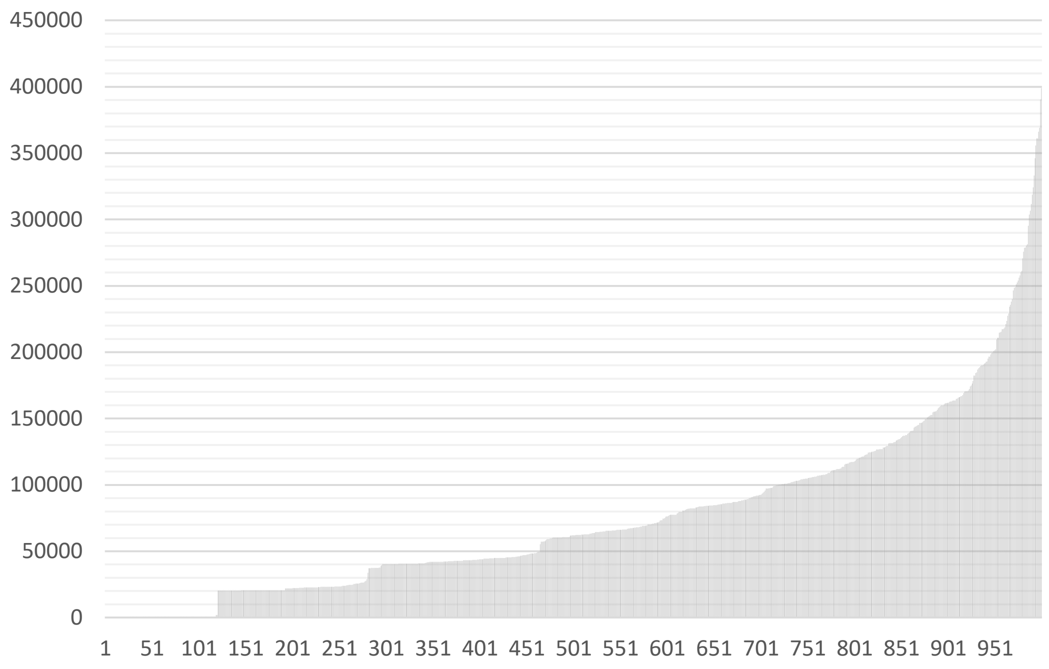
In the next plots we will see the number of ICache Raw misses from all benchmarks executed with Time randomized processor, we will be able to see the difference between them and compare them with the COTS.

- a2time



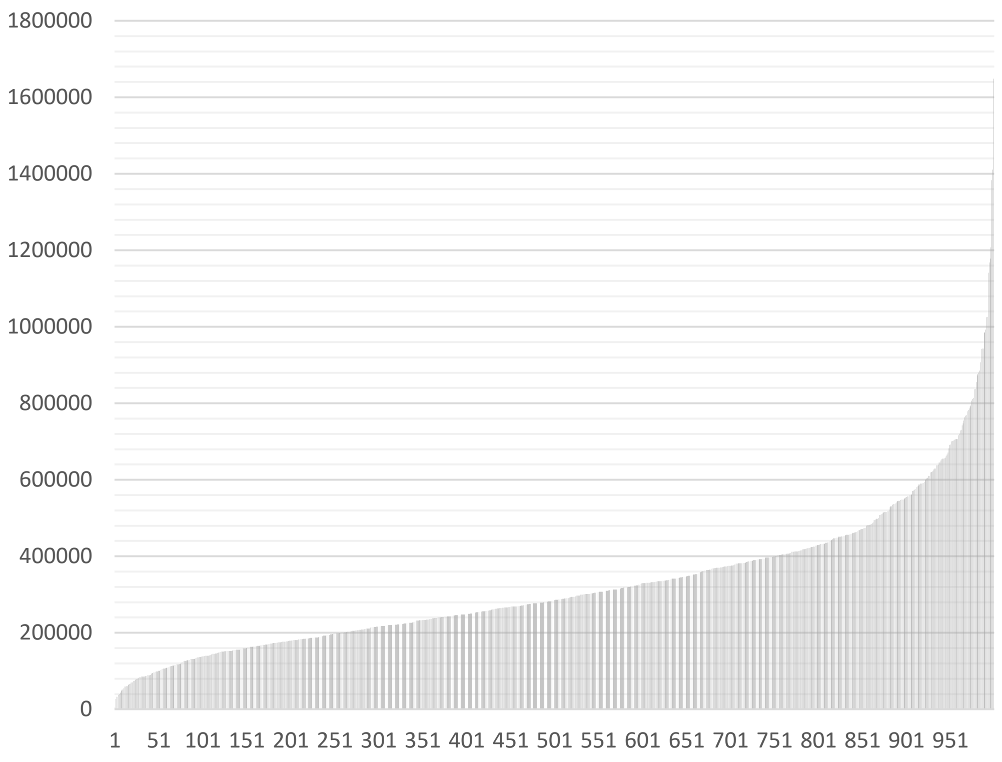
Plot 15: ICache raw misses of a2time for Time randomized processor

- aifirf



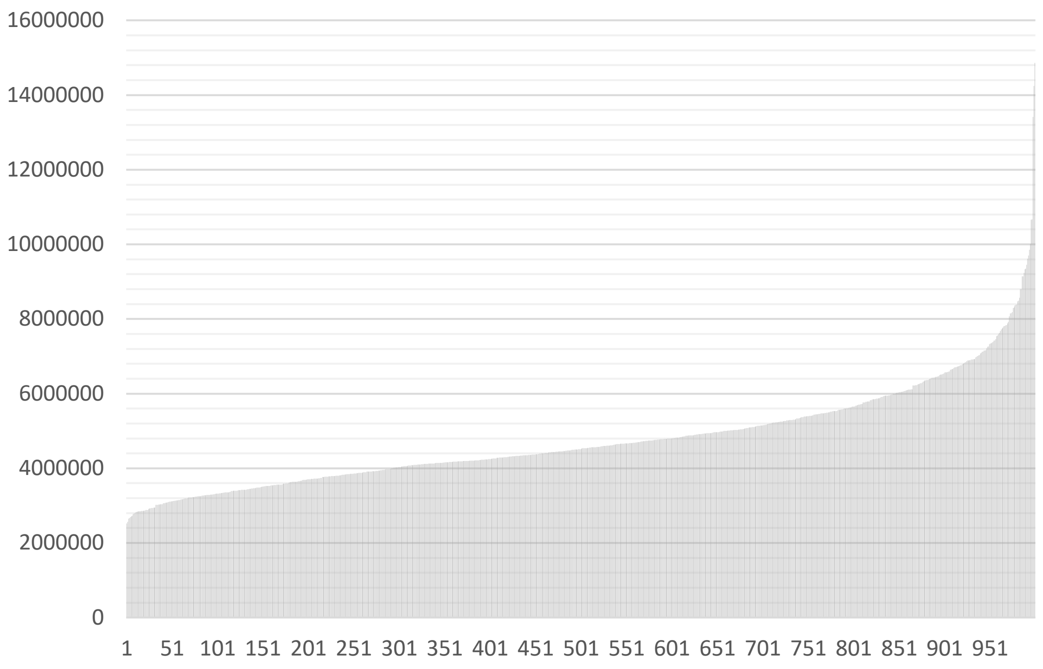
Plot 16: ICache raw misses of aifirf for Time randomized processor

- basefp



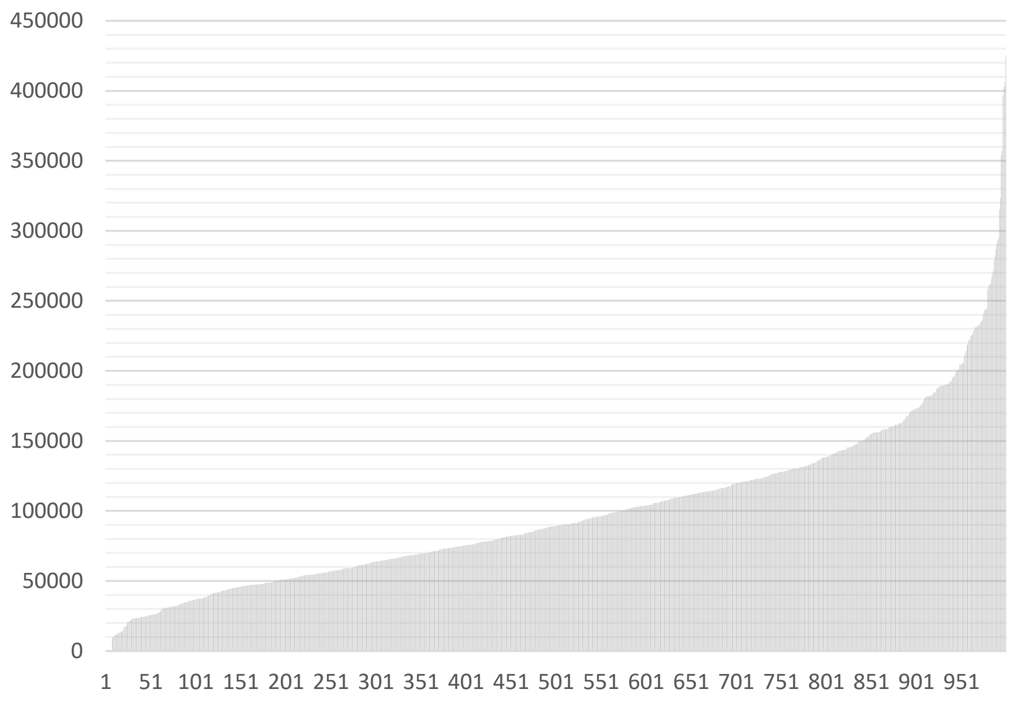
Plot 17: ICache raw misses of basefp for Time randomized processor

- bitmnp



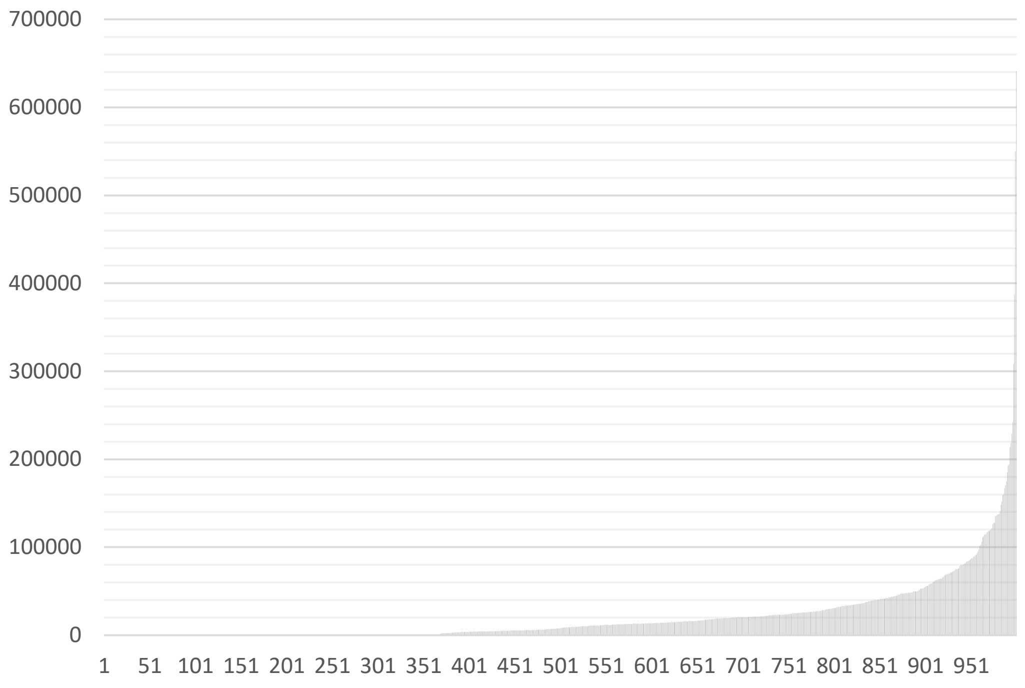
Plot 18: ICache raw misses of bitmnp for Time randomized processor

- cacheb



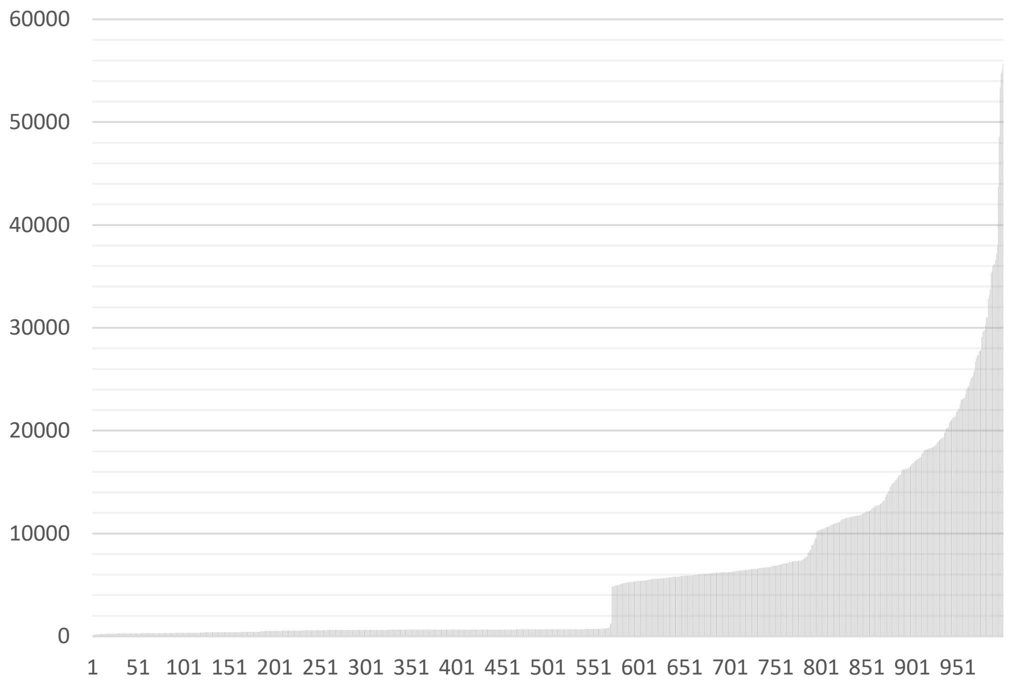
Plot 19: ICache raw misses of cacheb for Time randomized processor

- canrdr



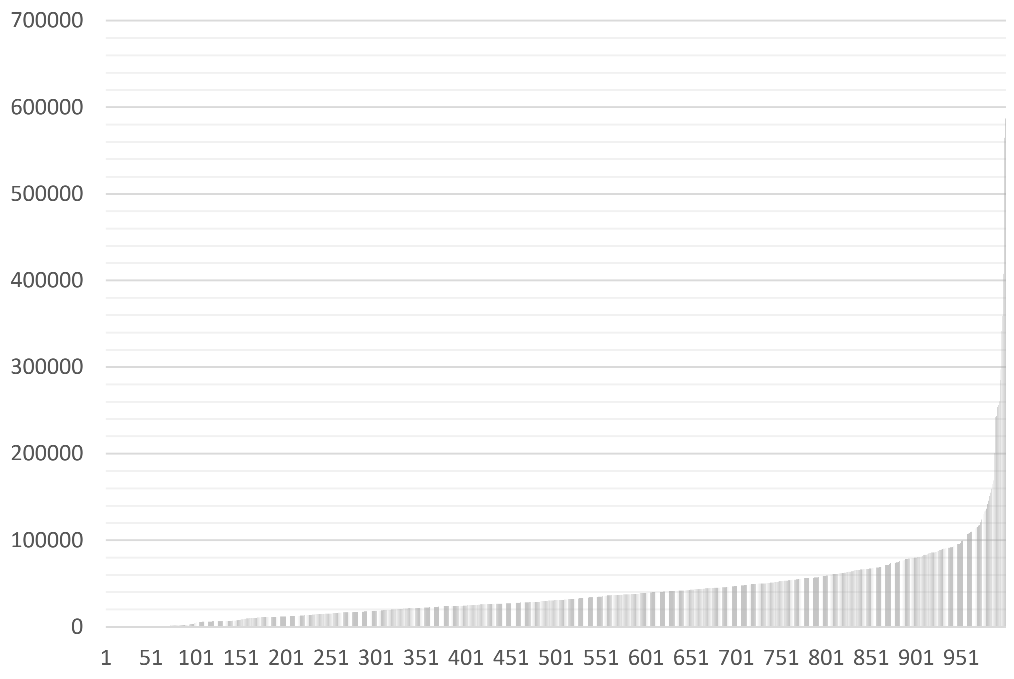
Plot 20: ICache raw misses of canrdr for Time randomized processor

- pntrch



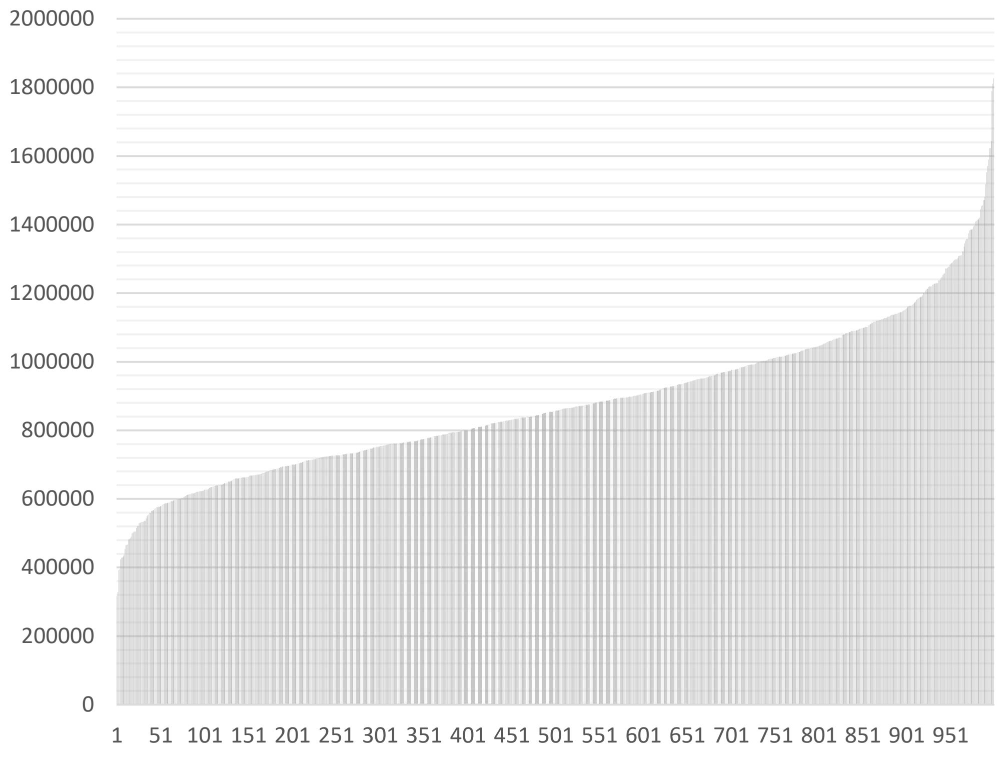
Plot 21: ICache raw misses of pntrch for Time randomized processor

- puwmod



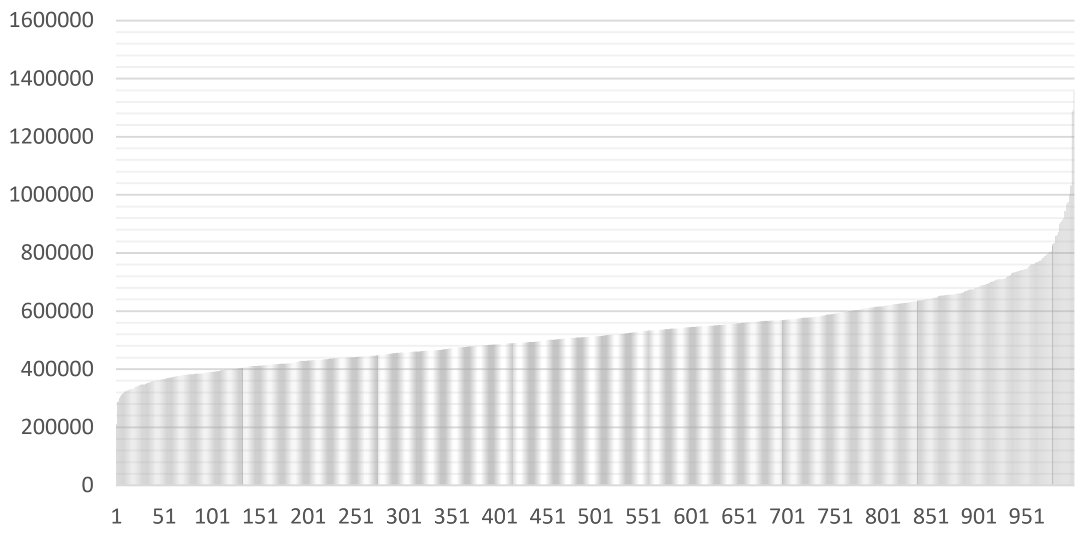
Plot 22: ICache raw misses of puwmod for Time randomized processor

- tblook



Plot 23: ICache raw misses of tblook for Time randomized processor

- ttsprk



Plot 24: ICache raw misses of ttsprk for Time randomized processor

To summarize the last plots, we can tell that all of them are more similar between them than the COTS plots mentioned before. On the other hand the shapes of all the Time randomized processors are very different from the normal plot except the tblook and ttsprk, and for that reason this two benchmarks will be easier for generating the performance signatures. So comparing with COTS implementation, most of the benchmarks will be easier for generating a performance signature with a COTS implementation but few of them like tblook and ttsprk are easier for generating performance signatures with Time randomized processors.

In table 13 we can see the coefficient of variation (%) for ICache misses for all the benchmarks tested with a Time randomized processor architecture.

Benchmark	Coefficient of variation (%)
a2time	0,462292508
aifirf	0,882816819
basefp	0,585893442
bitmnp	0,292120271
cacheb	0,592935916
canrdr	2,017133148
pntrch	1,465496714
puwmod	1,133303028
tblook	0,248526292
ttsprk	0,23753235

Table 15: Coefficients of variation (%) ICache miss for Time randomized processor

5.2.3 Evaluating performance signatures

In this section it will be evaluate the performance signatures from the statistics showed in the previous section.

5.2.3.1 Signature definition

After analyzing all the collected stats and as we have explained before, we will create the performance signature from the ICache raw misses.

In table 17 we can see the number of pathological cases that we can find depending on the number of misses for every benchmark with Time randomized processor implementation. For example 2X while tell the number of executions that overcome twice or more the mode, and the same for the fifth, tenth and hundredth.

benchmark	2X	5X	10X	100X
a2time	11	0	0	0
aifirf	882	880	880	573
basefp	221	7	0	0
bitmnp	39	0	0	0
cacheb	319	11	0	0
canrdr	632	631	597	154
pntrch	430	430	289	0
puwmod	223	21	7	0
tblook	3	0	0	0
ttsprk	5	0	0	0

Table 167: Pathological cases compared with mode of ICache Raw misses for Time randomized processor

In table 18 we can see the same as in table 17, but instead of using mode to filter we will be using the mean.

benchmark	2X	5X	10X	100X
a2time	10	0	0	0
aifirf	121	2	0	0
basefp	61	1	0	0
bitmnp	9	0	0	0
cacheb	56	0	0	0
canrdr	137	38	6	0
pntrch	180	25	1	0
puwmod	93	12	3	0
tblook	3	0	0	0
ttsprk	3	0	0	0

Table 17: Pathological cases compared with mean of ICache Raw misses for Time randomized processor

In the next two tables we can see the average of interruptions triggered for the false positives with mean and mode as point reference.

benchmark	2X	5X	10X	100X
a2time	1,21244857	0	0	0
aifirf	94,7938251	38,0016699	19,0008349	2,54385952
basefp	1,43704535	1,25937105	0	0
bitmnp	1,18436371	0	0	0
cacheb	1,43538803	1,21490897	0	0
canrdr	42,7337688	17,1193648	9,00322393	2,28823704
pntrch	9,54889362	3,81955745	2,40266366	0
puwmod	1,70989909	1,87747456	1,44321159	0
tblook	1,07323616	0	0	0
ttsprk	1,17933368	0	0	0

Table 18: Average of interruptions triggered for false positives with mode as a reference for Time randomized processor

benchmark	2X	5X	10X	100X
a2time	1,191382752	0	0	0
aifirf	1,409350358	1,04366646	0	0
basefp	1,29990305	1,018022468	0	0
bitmnp	1,19569138	0	0	0
cacheb	1,290891227	0	0	0
canrdr	2,247231926	1,686080723	1,803461933	0
pntrch	1,784796689	1,353203311	1,001457359	0
puwmod	1,675914454	1,676245347	1,275941424	0
tblook	1,027256397	0	0	0
ttsprk	1,237296808	0	0	0

Table 19: Average of interruptions triggered for false positives with mean as a reference for Time randomized processor

6 Conclusions

The main conclusions we draw from this work is that despite real-time workloads show little variability in IPC, execution time and even in cache accesses, misses and hits using performance signatures is not always a good option since sometimes we cannot discriminate the intrinsic innocuous variability from the performance variability caused by malicious software. However, in general the fact that real-time benchmarks have little variability is good to derive performance signatures.

However, some pathological cases can still occur and thus, causing some false positives to be triggered, this means that using this performance signatures there will be some good executions will be treated as a malicious which can be problematic since this can prevent real-time applications from completing executions in time.

We can choose a precision with our performance signatures that has the best relation between good executions treated as bad executions and vice versa.

On one hand we have create performance signatures using the mode of ICache misses as a reference in top of COTS processors. The pathological cases (false positives) in average for 2X are 20%, even so we can decrease this percentage using bigger performance signatures like 100X where the percentage is 7%. But we have to take into account that the false positives do only occur in 4 benchmarks only, for the other ones the false positives are 0 which means that for these benchmarks performance signatures are able to detect malicious attacks with a perfect match.

On the other hand we have used the mean as a reference to generate performance signatures. In this case we can see that the false positives are less than using mode, for example for a precision of 2X we have 5% of false positive.

We have also analyzed the suitability of Randomized platforms. The first problem we see is that for this type of workloads we again little variability due to cache conflicts in IPC but at the same time the amount of conflicts have increased significantly. The reason for having increased miss rates is the worse temporal and spatial locality properties of time-randomized caches. The good point is that the relative variability in this platforms is lower (Coefficient of variation).

The first difference we can see comparing the pathological cases tables between COTS and Time randomized processor is that in COTS we just have 4 benchmarks with pathological cases and for Time randomized processors we have pathological cases for all of them (regardless of the reference point that has been used).

Even so if we analyze more, we can see that the pathological cases in average that are false positives are more or less the same, for 2X (mode as a reference) the percentage will be around 30 % and for the 100X it will be around 7% (the same as COTS).

Also we can see that with mean as a reference we have less false positives (like in COTS implementation), for example for 2X is around 7%.

So, what we conclude that thought Time randomized processors offer more performance stability the loss of locality properties clashes with the good properties they offer being COTS platforms a better choice for performance signatures when using applications with little performance variability.

7 Glossary

COTS

Means Commercial off-the-shelf, and is the traditional implementation for a cache. In this implementation we can use different placement algorithms like FIFO.

Time-randomized

Random implementation for a cache. It consists on placing and replacing the data in the cache using randomized algorithms from memory direction, in this way we make sure that in bad optimized cases the accesses to cache are the different for each execution.

SPARC Simulator

Is a simulator that simulates with a high accuracy the 4-core NGMP processor, expected to be the target multicore platform for the next European Space Agency missions.

Workload

Quantitative amount of work that which carries out the execution of a benchmark.

EEMBC

Means Embedded Microprocessor Benchmark Consortium, is a non-profit, member-funded organization formed in 1997, focused on the creation of standard benchmarks for the hardware and software used in embedded systems. The goal of its members is to make EEMBC benchmarks an industry standard for evaluating the capabilities of embedded processors, compilers, and the associated embedded system implementations, according to objective, clearly defined, application-based criteria. EEMBC members may contribute to the development of benchmarks, vote at various stages before public distribution, and accelerate testing of their platforms through early access to benchmarks and associated specifications.

SoCLib

Is an open platform for virtual prototyping of multi-processors system on chip (MP-SoC).

ARVEI

Is a high-performance cluster from the UPC DAC (Architecture Computer Department). This cluster is used for research works as simulations, parallel works, intensive calculation, executions of a large number of virtual machines, etc.

NGMP

Means Next Generation Microprocessor, is a quad-core processor to be used in the future space missions of the ESA (European Space Agency).

TASA

Means Toolchain-Agnostic Static Software Randomization for Critical Real -Time System, and is a compiler given by the project directors that produces binaries of programs with randomized memory layout.

Performance signatures

Consist on the normal standards in form of variables or threshold that defines the correct behavior of a program.

Critical applications

Is any application, program or software that is essential to business operation or to an organization. Some examples can be the autonomous car systems, nuclear reactor safety system software, etc.

Coefficient of variation

Is a standardized measure of dispersion of a probability distribution or frequency distribution. It is often expressed as a percentage, and is defined as the ratio of the standard deviation σ to the mean μ.

Performance monitoring counters

Counters that can show real-time CPU variables like the cycle executed, hit/misses in cache, etc. and can be used to control all the aspects of a program execution.

ICache

Is the instructions cache that use the SPARC CPU.

DCache

Is the Data cache that use the SPARC CPU.

UCacheL2

Is a second level Cache that is accessed when we have miss in ICache or DCache.

Appendix A

Project planning

The estimated Project duration is about 4 months, from the end of January 2018 until the start of June 2018. As a disclaimer, since it is a project that depends in large extent on the analysis of unpredictable results, it must be pointed out that the initial planning could be updated during the evolution of the project.

Project analysis and design

The main objective of this phase is to make an accurate analysis of the project and develop the consequent design. On the one hand, in the project analysis it will be necessary to define and set the objectives, requirements, features and the use cases of our application. Furthermore, the state of art will be expanded and it is provided an analysis and evaluation of the different technologies used to the development. On the other hand, project design consists on creating the architecture of the software, i.e. sequence diagrams, database design, etc. That implies using all the knowledge acquired during the Bachelor Degree in Computer Engineering in order to make a high quality software.

Task description

In this section, we will try to describe the different tasks that make up this project ordered chronologically.

Project management

This task consists of the work that is developed for the GEP (Project Management) subject of the FIB. It will be constituted of 7 deliverables (subtasks), each of them will require a different time of dedication. In addition, every deliverable is divided in two parts, the documentation and the rubric. The deliverables are:

- Context and scope of the project (12.00 hours of dedication)
- Project planning (7.00 hours of dedication)
- Budget and sustainability (10.00 hours of dedication)
- Preliminary presentation (3.00 hours of dedication)
- Fold of conditions (12.50 hours of dedication)
- Final document (15.00 hours of dedication)
- Oral presentation PowerPoint (3.00 hours of dedication)

The resources needed to carry out these tasks are a computer (with internet access), Microsoft Office (Word and PowerPoint), Gantter, Racó of FIB and Atenea of UPC access and Google Drive.

Knowledge and understanding of the testing environment

The first step and essential task that we must perform before starting with the collection and analysis of data, will be to familiarize with the work environment and understand the operation of the tools we need use.

Environment familiarization

Ubuntu is the OS distribution that will be used to perform the entire workload of the project except for the first task mentioned above (this task will be performed using a Windows OS distribution). Thanks to the different subjects attended in the FIB in which study and use of Ubuntu OS has been done, I have not needed to spend too much time. The environment is provided in a virtual machine with the basic tools installed, to facilitate the project's bootstrap time.

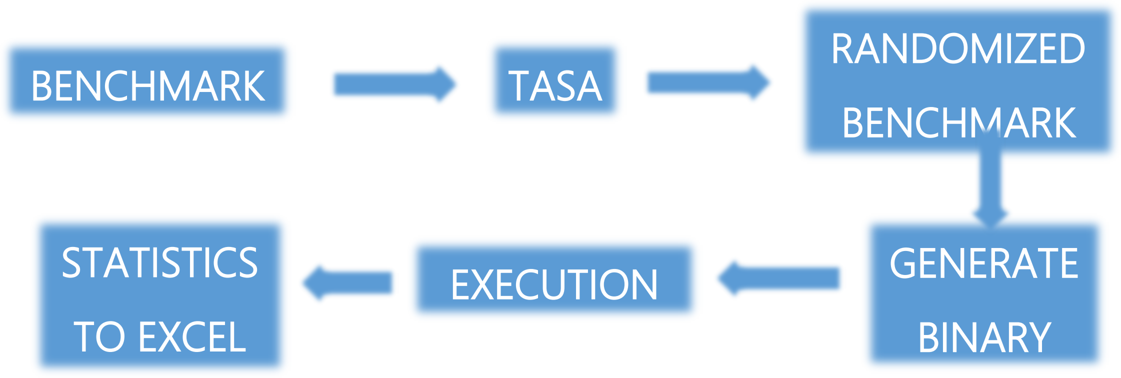
Understanding of the operation of the tools

TASA is the source-to-source compiler that we will be used to generate equal binaries but with different type of code randomized modifications, depending of the flags used.

SPARC simulator is the program that will be used to realize the first CPU behaviour study. Since there have to make lot of executions with this simulator it is needed a knowledge of it to try to perform the maximum number of executions in the shortest possible time, due to the limited deadline to finish the project as mentioned before.

Implementation of automation scripts

Since this is a project to study the performance of a CPU, it requires to collect lot of statistics. For that reason, it will be needed to generate lot of randomized benchmarks and make lot of executions (around 100.000 executions for every CPU type) and collect the statistics of all of them.



Due to the limited time we have, it's a must to create optimized scripts to automatize this process. It has been implemented 4 scripts:

- Randomized benchmarks generator: this script is very interactive and it is in charge of generate the randomized benchmarks from each benchmarks. The parameters that you can set are the benchmark (one or all), the flags, and the number of randomized benchmark. Run this script it takes 1.2 seconds per number of randomized benchmark.
- Binary generator: this script it's also very interactive and it is in charge of compile the benchmarks for SPARC architecture and generate the correspondent binary. The parameter you can set is the benchmarks from which you want to generate the binary. Run this script it takes 0.1 seconds per benchmark
- Statistics generator: this script is in charge of generate all the statistics from the execution of one benchmark and his equal randomized benchmarks. The parameter is the benchmarks that you want to execute. Run this script it takes 6 seconds per execution.
- XLS generator: this script is in charge of convert all the statistics of one execution to an XLS document. The parameter is the statistics you want to convert to XLS document.

Analysis of statistics

Once we have all the statistics in an XLS spreadsheet, we will generate some plots to see which the behaviour of the CPU is and we will compare the results using different flags in cache. We will focus on several statistics:

- Execution time
- Used memory
- CPU usage
- LR1 cache missed
- LR2 cache missed

In this task we will have to take into consideration feedback from the directors. To perform this task frequent communication with the directors it will be needed, because I don't have enough knowledge about the usual CPU performance.

Resources

To develop correctly this project, it will be used the next resources.

Hardaware:

- Personal computer (Own design and build)
- Remote BSC machine (6 CPUs)
- Remote BSC Cluster (100 CPUs)

Software:

- Linux Virtual Machine
- VirtualBox
- EEMBC Benchmarks
- NGMP simulator
- MobaXterm
- TASA
- XSL library
- OpenOffice
- Microsoft Office 365
- Windows 10 professional

Schedule

In the next illustration we will be able to see the initial Gantt diagram of the task designed for this thesis. This diagram has changed several as the project progressed

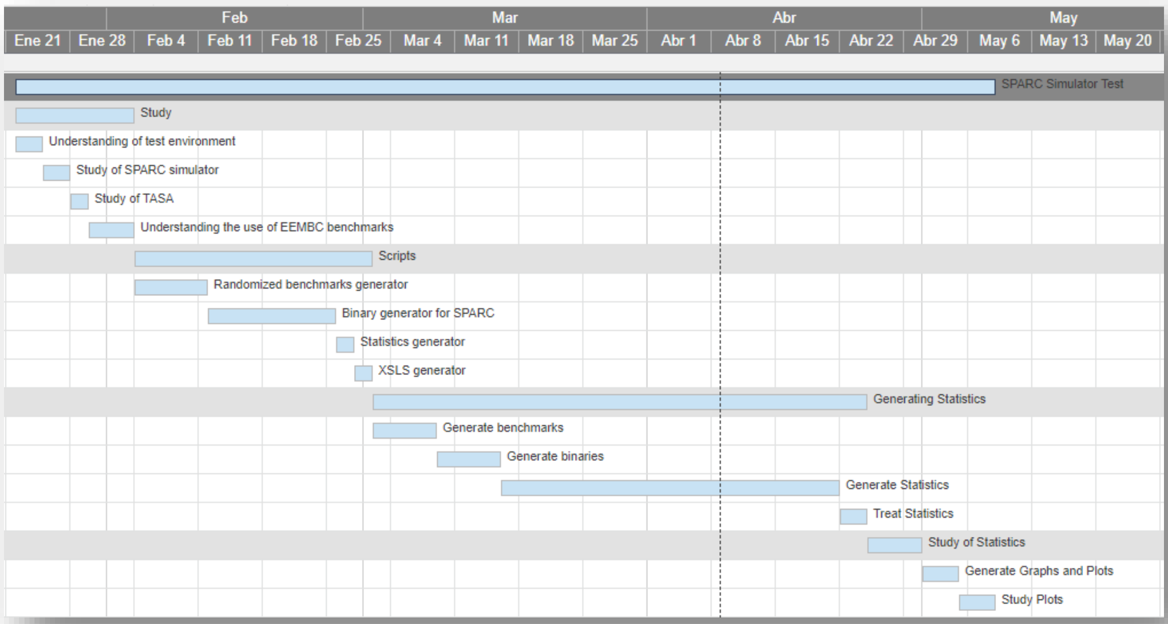


Illustration 20: Initial Gantt diagram

Appendix B

Sustainability

In this section we are going to evaluate the sustainability of our project in three different areas: economic area, social area and environmental area.

Economic sustainability

In this document we already can find an assessment of the costs of our project, taking into account hardware, software and human resources.

The cost stated in the Total budget section of this document could be the only spent in the project, since our project is a study that can be realized using OpenSource applications. However, we can also use non OpenSource applications with more features and with a best easy-use.

It would be difficult to do a similar project with a lower cost. Due to the most of the costs are from human resources. Despite this, we could reduce the hardware budget by using a cheaper computer and without using the remotes cluster and machine, but this can affect directly to the computation times, and we will need more time to develop the project. Furthermore, we can also decrease the software budget just using cheaper Office and Windows versions, but given the fact that we already have these versions we did not buy previous versions.

Social sustainability

The study that we are going to develop in this project is going to be used for the CPU builders or for cybersecurity of CPU companies, since our goal is use CPU Performance Signatures for Security Attacks Detection.

This study can help in the improvements of antivirus. In addition, it we can be very helpful for all the companies to detect cybersecurity attacks easily and while a malicious application without producing damage in the CPU.

Environment sustainability

The resources used in the project have been detailed in the Budget estimation section.

Product	Power	Use	Consumed energy	Total estimated CO_2
Personal computer	500 W	600 h	300 kWh	115,50 Kg of CO_2
Total estimated	500 W	600 h	300 kWh	115,50 Kg of CO2

Table 20: Energy estimated costs

In Table 5, an estimation of project energy cost is provided.

The only resources that can affect to environment and we use in this project will be the Personal computer (use), the energy spent and the paper used to print the documentation. Knowing that, we can estimate the energy spent developing the project.

It is in fact a high amount of energy, but since we need to make extensive use of a computer there is no way to reduce it.

Sustainability Matrix

Type	Project Development	Exploitation
Environmental	10	0
Economic	7	5
Social	10	8

Table 21: Sustainability Matrix

8 References

[1] Carles Hernández, Jaume Abella, Francisco J. Cazorla, Alen Bardizbanyan, Jan Andersson, Fabrice Cros, Franck Wartel: Design and Implementation of a Time Predictable Processor: Evaluation With a Space Case Study. ECRTS 2017: 16:1-16:23

[2] Mikel Fernández, David Morales, Leonidas Kosmidis, Alen Bardizbanyan, Ian Broster, Carles Hernández, Eduardo Quiñones, Jaume Abella, Francisco J. Cazorla, Paulo Machado, Luca Fossati, Probabilistic timing analysis on time-randomized platforms for the space domain. DATE 2017: 738-739

[3] Javier Jalle, Mikel Fernandez, Jaume Abella, Jan Andersson, Mathieu Patte, Luca Fossati, Marco Zulianello and Francisco J. Cazorla, Contention-aware performance monitoring counter support for real-time MPSoCs, IEEE Symposium on Industrial Embedded Systems (SIES) 2016

[4] Leonidas Kosmidis, Roberto Vargas, David Morales, Eduardo Quiñones, Jaume Abella, Francisco J. Cazorla, TASA: Toolchain-Agnostic Static Software Randomisation for Critical Real-Time Systems. ICCAD 2016: 59

[5] J. Poovey, Characterization of the EEMBC Benchmark Suite. North Carolina State University, 2007.

[6] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, Salvatore Stolfo, "On the Feasibility of Online Malware Detection with Performance Counters", International Symposium on Computer Architecture (ISCA) 2013

[7] Sanjeev Das, Hao Xiao, Yang Liu, Wei Zhang, Online malware defense using attack behavior model. ISCAS 2016: 1322-1325

[8] Charlie Curtsinger, Emery D. Berger, STABILIZER: statistically sound performance evaluation. ASPLOS 2013: 219-228

[9] J. Poovey, Characterization of the EEMBC Benchmark Suite. North Carolina State University, 2007.

[10] Carles Hernández, Jaume Abella, Andrea Gianarro, Jan Andersson, Francisco J. Cazorla: Random modulo: a new processor cache design for real-time critical systems. DAC 2016: 29:1-29:6