POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Virtual Tool-boxing for Robust Management of Cross-layer Heterogeneity in Complex Cyber-physical Systems

Supervisors
prof. Stefano Di Carlo
prof. Alessandro Savino

Candidate
Simone Dutto

October 2020

POLITECNICO DI TORINO

Abstract

Computer Engineering
Master's Degree

Virtual Tool-boxing for Robust Management of Cross-layer Heterogeneity in Complex Cyber-physical Systems

Nowadays, due to technology enhancement faults are increasingly compromising all kinds of computing machines, from servers to embedded systems. Many methods have been proposed in these years, however, machine learning brought many possibilities to empower faults detection exploiting hardware metrics inspection, and it is now possible to explore the opportunity of avoiding the use of heavy software techniques or product-specific errors reporting mechanisms.
In this thesis, data analysis will be performed on several datasets collected by many simulated runs, with and without faults injection. The final goal of this work is to find the best machine learning model to cope with the task and ultimately to build an initial implementation of a monitoring tool, able to detect faults process-wise using pre-trained models on hardware metrics extracted by a kernel module.

Contents

Chapter 1

General Introduction

«Error monitoring is a critical procedure for most computing systems, varying from HPC to embedded systems domains.» [9]
Hard errors always have been an issue faced during the design of a circuit. These types of faults can be caused either by project bugs or materials ware out. However, technology scaling (increasing transistors density, low energy consumption requirements) has been discovered being a key factor in highering the rate of soft errors in hardware devices. [3]
This situation puts in the hand of researchers the goal of finding a way to cope with errors: for example, a hardware protection technique fairly known and used in lots of cache memories is Error Correction Codes. In addition to hardware mechanism also software error detection has been developed, many of these works are based on redundancy to ensure error detection. In fact, these techniques impact the performance of systems. Just to give an idea, according to Mukherjee et al. [32] his technique called RMT (*redundant multithreading*), which consists of detecting a fault by running two identical copies of the same program as independent threads and compares their outputs, has a degrade percentage on performance on commercial microprocessors of 30%.
After that there has been a lot of interest in creating fault monitors built as Linux kernel extension: HealthLog [9] is one of the most comprehensive and powerful monitoring tools, it is based on errors reporting from hardware and a kernel module able to handle these report messages and react accordingly to user specifications.
All of these methods have some limitations: first of all, they require specific hardware supports, they either work at the hardware level directly or exploiting hardware error reporting. This is, indeed, the safest way to detect errors

but it is dependent on hardware manufacturers to add reporting mechanisms for this or that specific error. Also, hardware techniques are not process-wise, because they keep tracks agnostically of errors in memory structures. Exactly for these reasons, it is interesting to explore faults detection using hardware metrics monitoring (not error reporting) and machine learning, to enable flexibility and precision trying to keep the mechanism as lighter as it is possible both hardware and software-wise.
The ultimate goal of my thesis is to discuss what can be done in terms of machine learning models to detect faults and finally to give a primal implementation of a tool to monitor processes in a Linux environment.
Building a machine learning-based faults monitor is composed of three parts: the first is collecting several datasets to work on, the second is investigating different models to cope with faults detection, and lastly designing an infrastructure that integrates metrics inspection and fault detection.
In the following chapters, a fault is considered to be an inspected behavior in a processor structure (i.e. Program Counter, Register File, Arithmetic and Logic Unit).

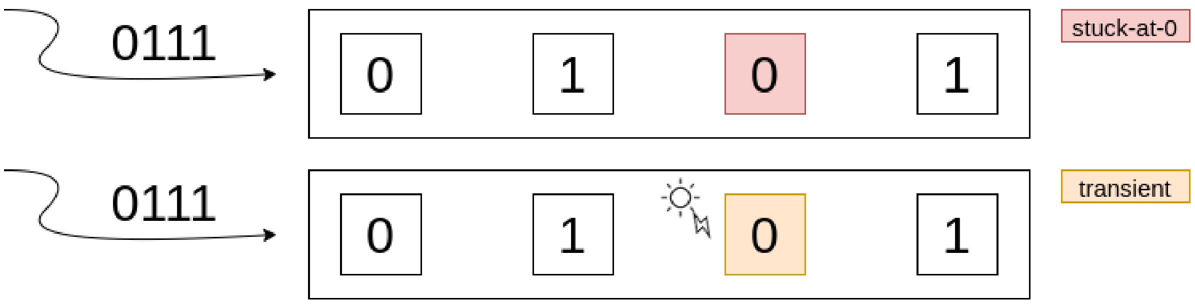


Figure 1.1. Two types of fault in a Program Counter: the address that will be used is not the one that was written. The first fault is caused by a permanent 0 in the memory, the second is caused by a bit flip due to a cosmic ray

Of course, during the execution of a program, there are a lot of memory or address references and these faults can severely affect a running system [11]. Unfortunately, faults are difficult to be injected arbitrarily into a processor for their unpredictable nature. For example, «Given 1000 – 1500 FIT/Megabit on a device with 48Kbyte of RAM and an operating lifetime of 104 hours, one out of every 175 to 250 devices should be expected to see a RAM bit

flip at some point in their operating lifetime» [23]. It is really difficult to observe the effects of a rare fault like this. For this reason, the gem5 simulator [5] in combination with FIMSIM has been used. Gem5 is a simulator able to simulate full computer system architecture, run a Linux OS, and extract hardware metrics. FIMSIM [51] is an add-on to gem5 that enables different types of fault injections. It is very important to notice not all the hardware metrics extracted by the simulations exist in current systems.
The first chapter will explain how the simulated system is created and how it is handled by the simulator implementation-wise. Later the FIMSIM functionality will be inspected and explained alongside the different types of faults and their respective effects on the system. Finally, the metrics generated from different simulations are explained in the context of models creation.

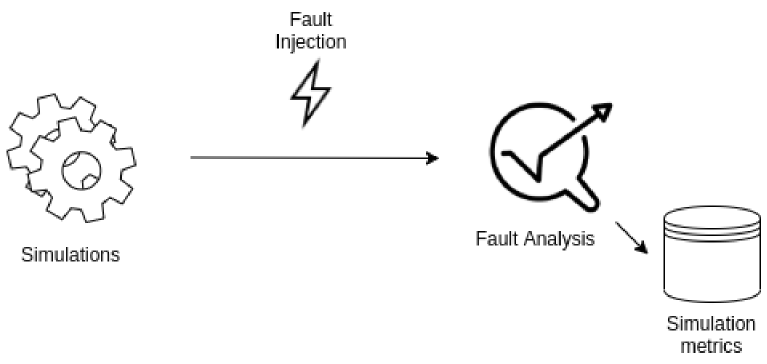


Figure 1.2. Metrics collection

After having collected the dataset, in the second chapter topics covered are three: data analysis, models inspection, and experiments results.
Data analysis consists of extracting the data from the simulation metrics and cleaning up the stats. After the extraction a features selection is analyzed: preparing the features is key for the final application. Both because the number of metrics inspected highly impacts the performances of a system and also because a different number of features implies different models.
At least, a review of related works on fault detection is investigated. it must be pointed out the simulations datasets are created for this work only and they are not part of a benchmark suite for faults detection, as S5 Yahoo Anomaly Dataset [48]. So, there is no literature regarding models applied to the datasets taken into consideration in this thesis. Hence, a different approach has to be adopted.

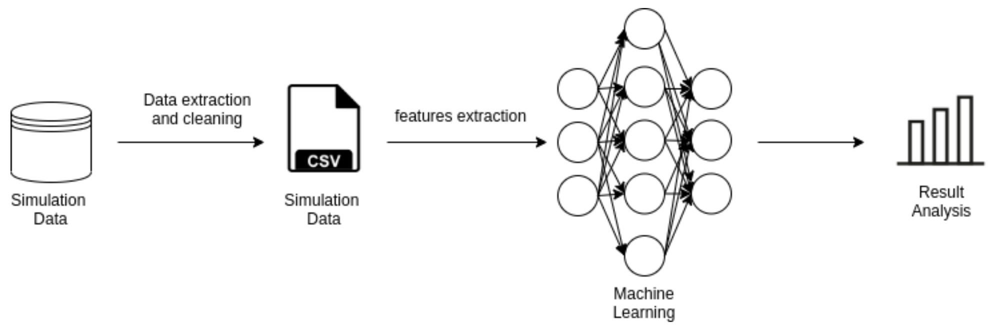


Figure 1.3. Fault detection Model

Different architecture and domain in the fault detection research field are analyzed and taken as inspiration for models used later on the actual datasets. At last, each model is tested and performances are compared. Again, the goal is not to find the best model to beat the accuracy on a competition dataset but to find the best model to perform on the task and propose ideas to further investigate fault detection in operating systems.
The third chapter is about implementing the whole system on a current Linux distribution. During this chapter the basic functioning of a Linux kernel will be explained and how it is possible to interact with it exploiting the kernel API to build complex systems through a kernel module. In addition to that, PMCTrack [36], a hardware metric extractor, will be analyzed in depth. This module is the core of the chapter because from this work it is possible to extract important concepts to build the actual monitor. However, it is important to remind what it has been said at the start: most of the metrics collected are not present in nowadays system. As a consequence, the system implemented is a proof-of-concept of what could be achieved with better hardware support.
The system designed, Figure 4.3, is based on expanding PMCTrack to use the metric extractor to evaluate a process to decide whether or not there has been a fault during its execution.
The system decouples the machine learning evaluation from the kernel for simplicity reason: floating-point operations are not permitted kernel level and ML libraries are based on them, and also because it is common to make this kind of operations on machines with special hardware (GPU) and it is not given each machine with a fault monitor has that hardware.

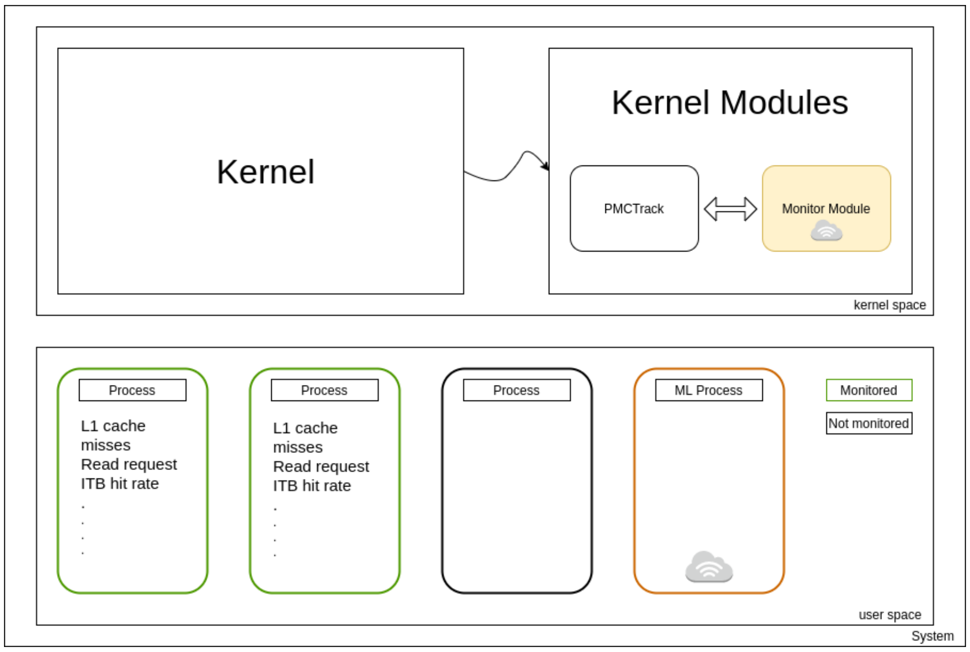


Figure 1.4. Final system: the monitor kernel module is used to handle the whole process, the PMCTrack module to extract the metrics then forwarded to a Machine Learning process running inside (or outside) the machine that gives the evaluation back to the monitor module

The last chapter will have final thoughts and mostly what the author considers to be the best ways to continue the thesis and expand what it has been done.

Chapter 2

Simulations Environment

2.1 Gem5

Gem5 [5] is a configurable simulation framework, it is a combined work of several academics and companies, including AMD, ARM, HP, MIPS, Princeton, MIT, and the Universities of Michigan, Texas, and Wisconsin. It has been used in thousands of publications, it is community-led and released under a BSD license.
The main aspects of gem5 are flexibility, availability, and encouraging developer work to expand the simulator.

2.1.1 Gem5 Architecture

Understanding gem5 inner implementation is very valuable in order to set up simulations.
Gem5 is written in C++ and Python and it is based on 4 pillars:

- *pervasive object orientation:* all components are SimObject and share common behavior for configuration, initialization, statistics, and serialization. Each SimObject has 2 classes: one in C++ and the mutual in Python. Python classes are used to compose the initialization scripts and exposing a common interface and the C++ ones are used to implement the specific behaviors during the simulation.

- *Python Implementation:* Python as mentioned before is used for the startup scripts and simplifies the configuration of each object.

- *Domain-specific languages:* in some cases we need the flexibility to explain very domain-specific behavior of components. For this reason, gem5 supports DSLs. To give an example, it supports SLICC, which is a DSL to define cache coherence protocols.

- *standard interfaces:* they come after the object-oriented design. For example, each object has a port interface, through this the SimObject communicates with other SimObjects. This gives maximum flexibility, decoupling implementation of the SimObject from the communication with others.

2.1.2 Gem5 Simulation Parameters

CPU Model. It is possible to simulate two families of processors. The first is based on SimpleCpu implementation. This class implements in-order not-pipelined models, defining functions to handle interrupts, setting up fetch requests, advancing PC, and implementing the ExecContext interface (responsible for RW operation on memory, ALU, PC, Cache). In this family, there are two distinct subclasses, AtomicSimpleCPU and TimingSimpleCPU. They are different because the first one uses atomic memory access and the latter timing memory access. One is implemented with a one-way message without a real sender/receiver queue and response time is just approximated, the other is more detailed and is implemented with 2 one-way messages. This way the simulation reflects better the realistic timing of memory access.
The second family of processors is more complex, and it comprehends In-Order CPU and O3 CPU. The first is a configurable in-order pipelined CPU. The second one is the most complex CPU, it simulates out-of-order pipelined CPU, with reorder buffer. This last one permits to simulate superscalar architectures and hardware multi-threading.

Memory System. Memory system is implemented through MemObject implementing a memory structure (CacheObject for example) and ports to communicate with other SimObjects through a master/slave interface. Usually, each master port is connected to an interconnect component, like a bus or bridge.
Once connected, it is possible to send messages (a Request or a Packet). This is where most of the statistics not usually present in real systems are extracted because each Request/Packet object can be expanded with new attributes (ex. time request was created, ID of CPU/thread that causes the request).
To better understand from where statistics are coming from, in figure 2.1 is

presented the flow of a Request Object with a 2-level cache memory architecture.

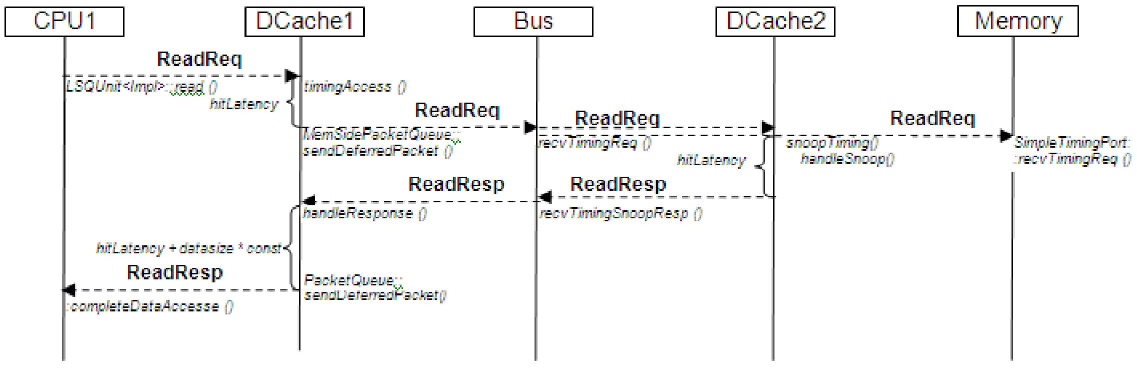


Figure 2.1. Request message flow from CPU to principal memory and back. We can notice how it is organized to interact with each SimObject. Figure from [5].

At last, gem5 includes two different types of memory system coherence: classic and Ruby. The classic cache is a non-blocking cache with MSHR (miss status holding register) and WB (Write Buffer) for read and write misses. Substantially with Ruby it is possible to declare MemObject with complex cache coherence protocol, topology and other custom parameters.

Execution Mode. There are two execution modes in gem5. The first one is Full Simulation, it simulates a bare-metal environment running a Linux Kernel image. It is the most complete and accurate simulation. The second one is System-call Emulation, in this mode whenever there is a system-call is trapped and passed to the host running gem5.

ISA. With gem5 is possible to simulate different Instruction Set Architecture without having to specialize each Object for each ISA. This is achieved by a ISA DSL, which unifies the decoding of binary instructing. It exploits a C++ base class and derived classes for each ISA which override methods like *execute()* to implement opcodes (ex. ADD, SUB).

Devices. Additionally is possible to add any kind of I/O device(DMA, NIC), but if it is used in FS mode it is needed to having working drivers.

2.1.3 Configuration Script

Configuration scripts are really easy to set up thanks to Python and SimObject abstraction.

The following code is a stripped-down version of the actual script that has been executed for the final simulations, it is simplified to avoid showing details about x86 specific implementation and other build options. In a few words in this file, this script creates the MySystem object, this object will be used later to execute simulations.

```
# Set up the clock domain and the voltage domain
self.clk_domain = SrcClockDomain()
self.clk_domain.clock = '3GHz'
self.clk_domain.voltage_domain = VoltageDomain()

mem_size = '512MB'
self.mem_ranges = [AddrRange(mem_size),
                   AddrRange(0xC0000000, size=0x100000), # For I/O
                   ]

# Create the main memory bus
# This connects to main memory
self.membus = SystemXBar()
self.membus.badaddr_responder = BadAddr()
self.membus.default = self.membus.badaddr_responder.pio

# Set up the system port for functional access from the simulator
self.system_port = self.membus.slave

# This will initialize most of the x86-specific system parameters
# This includes things like the I/O, multiprocessor support, BIOS...
x86.init_fs(self, self.membus)

# Change this path to point to the kernel you want to use
# Kernel from http://www.m5sim.org/dist/current/x86/x86-system.tar.bz2
self.kernel = 'binaries/x86_64-vmlinux-2.6.22.9'

# Options specified on the kernel command line
boot_options = ['earlyprintk=ttyS0', 'console=ttyS0', 'lpj=7999923',
                'root=/dev/hda1']
self.boot_osflags = ' '.join(boot_options)

self.setDiskImage('disks/linux-x86.img')

```

```
# Create the CPU for our system.
self.cpu = AtomicSimpleCPU()
self.mem_mode = 'atomic'

# Create the cache heirarchy for the system.
self.cpu.icache = L1ICache(self._opts)
self.cpu.dcache = L1DCache(self._opts)

# Connect the instruction, data, and MMU caches to the CPU
self.cpu.icache.connectCPU(self.cpu)
self.cpu.dcache.connectCPU(self.cpu)

# Hook the CPU ports up to the membus
self.cpu.icache.connectBus(self.membus)
self.cpu.dcache.connectBus(self.membus)

# Connect the CPU TLBs directly to the mem.
self.cpu.itb.walker.port = self.membus.slave
self.cpu.dtb.walker.port = self.membus.slave

# Create the memory controller for the sytem
self.mem_cntrl = DDR3_1600_x64(range = self.mem_ranges[0],
                               port = self.membus.master)

# Set up the interrupt controllers for the system (x86 specific)
self.cpu.interrupts[0].pio = self.membus.master
self.cpu.interrupts[0].int_master = self.membus.slave
self.cpu.interrupts[0].int_slave = self.membus.master
```

Listing 2.1. Configuration Script for MySystem: it contains definition and connections between different SimObjects

This script creates a system with single AtomicSimpleCpu with 2-level cache with x86 ISA.
Before continuing it is important to introduce briefly what are some SimObjects instantiated that were not explain before:

- **Bus.** The Bus Object has multiple slave port interface, and it is used to connect multiple MemObject. It works in the exact same way of a physical bus, and it transfers requests between the different components.
- **Memory controller.** The memory controller is an Object to handle data flowing from and to the principal memory of the system.
- **APIC.** It is initialized in the *x86.init_fs()* function and it is an interrupt controller, in charge of handling interrupt requests.

2.1.4 Simulation

After having defined the system it is time to start the simulations. It is simple and it provides some additional functionalities to custom the runs.

```python
# create the system we are going to simulate
system = MySystem(opts)
# set up the root SimObject and start the simulation
root = Root(full_system = True, system = system)
# instantiate all of the objects we've created above
m5.instantiate()
# start from a cherckpoint
m5.checkpoint(joinpath(cptdir, "cpt.%d"))
print "Running the simulation"
exit_event = m5.simulate(num_tick)
print 'Exiting @ tick %i because %s' % (m5.curTick(),
                                         exit_event.getCause())
```

Listing 2.2. Simulation script to estabilish MySystem and run it

As it is written in the code above, it is possible to start the simulation from an arbitrary checkpoint if there is the architectural state dump file and simulate for a certain number of ticks. After the simulation is finished, the current architectural state of the simulation is saved in a *.cpt* file and a file full of statistics about each Object simulated is created.

2.2 FIMSIM

FIMSIM [51] is a fault injection infrastructure built upon gem5. It is used to inject different types of faults at an arbitrary tick in multiple components.

2.2.1 FIMSIM basic functioning

Gem5 was chosen because it has 3 key properties to make a fault injector beneficial.

- Possibility to simulate a full system, so effects of faults can be observed both on OS and applications.
- Checkpointing mechanism and deterministic simulations: so it is possible to start whenever we want simulations(after booting the system for example) and it is guaranteed to have the same results with the same conditions.

- Command lines option: in gem5 basically everything can be modified by a command-line option, so it is easy to start different simulations without having to touch a line of code

- gem5 is object-oriented: the authors modified the *nextCycle()* function in the System Object to support fault injection without interfering with other parts of the code.

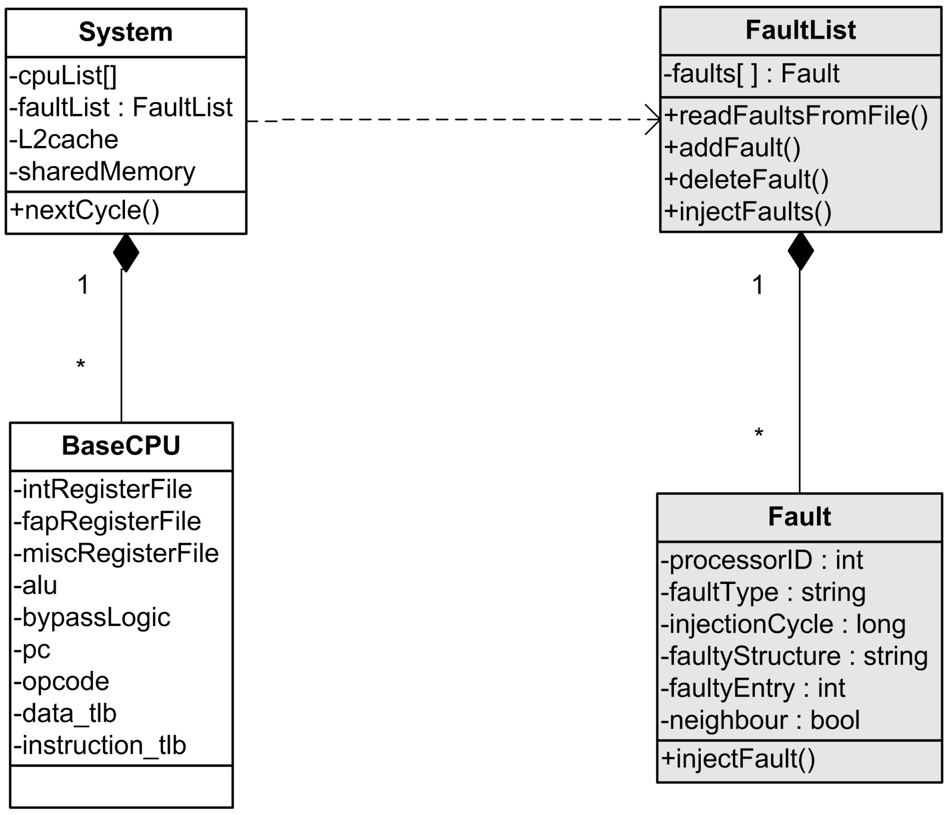


Figure 2.2. FIMSIM functioning: shaded boxes are objects added upon gem5. Figure from [51].

2.2.2 FIMSIM faults injection

Faults Definition The injection list in the figure 3.6 it is initialized at the start of the simulation, it just take a list of parameters for each fault:

- **Cpu Id.** CPU to inject the fault on.

- **FaultType and faultyStructure.** In semiconductors there are 3 types of faults.
  The first is *transient*, it is a bit flip due to some radiation event or power supply noise. These errors are nondestructive and they are restored after a write operation. However, they can lead to major problems [4].
  For example, if a bit flips in the Program Counter can cause a segmentation fault at best or even crash the OS.
  The second fault type is *permanent.* Permanents faults are indeed destructive and they cannot be reversed, the structure must be disconnected and then substituted. This fault can be divided in *stuck-at-0* and *stuck-at-1.* Stuck-at-1 tends to be more harmful because 0 is more common in structure, so a fixed 1 sooner or later is gonna cause some serious problem.
  The last type is *intermittent*, these faults are usually caused by wear-out, voltage, and temperature fluctuations and they are a burst or faults that last from some cycle to some seconds. They are simulated in FIMSIM via several subsequent stuck-at faults.
  All of these faults can be *singular* or *multi-bit.* Due to the shrinking size of the transistors, it is common that irradiations strike two bits, for example.
  All these types of faults can be injected in several structures: ALU, PC, intRF, ALU.

- **FaultyLine and bitPosition.** Each fault can be injected at any lines in a structure and in an arbitrary position. It is possible to use -1 in the definition string to randomize the position.

- **FastForwardCycle and faultCycle.** Fast forward cycle permits to set an initial cycle to start to inject fault. It is used to avoid injecting faults at boot time but during application execution. FaultCycle is the initial cycle to start to inject. A common combination used is: 30000 on fastForwardCycle, so the system can boot, and then -1 at faultCycle, so the start of the fault during an application execution is random but the system can boot safely.

- **Intermittent cycle.** Intermittent cycle defines the period between burst of faults in intermittent fault type

- **NeighboorFault and directionFault.** NeighboorFault defines the number of bit in multi-bit fault, directionFault the direction, it can be either vertical or horizontal.

Faults Results A fault can result in 3 categories: benign, error, and crash. The fault is spotted comparing the checkpoint in the golden run (run without faults injected) and the checkpoint with the fault injected.
Benign is when the fault has not produced any effects, so the two checkpoints are equals. E.g. this happens when a registry is modified by a transient fault but before it is read it is written, so the fault is masked.
Error arises when the two checkpoints are different. This implies there was a change in the architectural state because the check is independent of the output of the applications.
Crash occurs when the application crashes before the checkpoint. Unluckily this causes the loss of data and these simulations need to be discarded from the analysis. Nonetheless, crashes are not so interesting to analyze because they are already statically checked by hardware watchdogs.

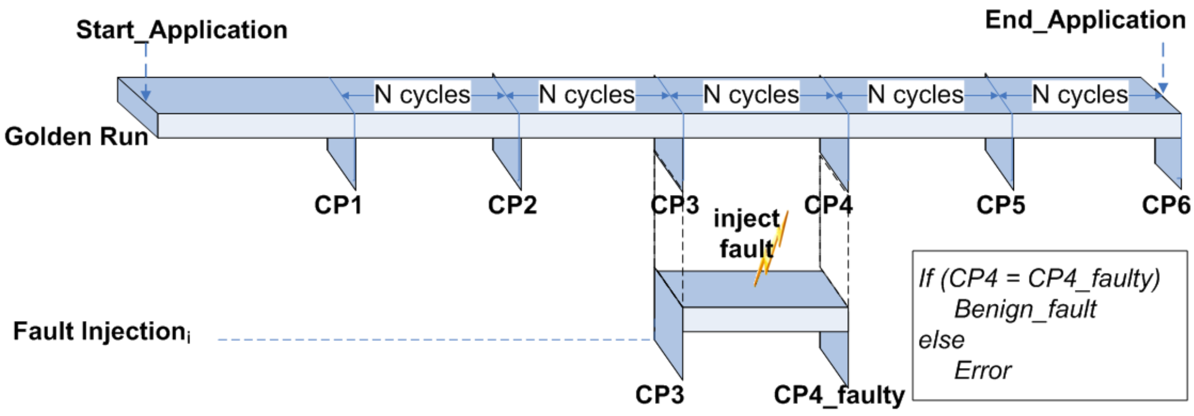


Figure 2.3. FIMSIM functioning: golden and faulty run. Figure from [51].

2.3 Data Collection

For the analysis data were collected running multiple simulations with 2 types of faults on a vanilla Linux kernel executing 4 binaries.

2.3.1 Faults Injection

Faults definition. Faults injected types are permanent (both stuck-at-0 and stuck-at-1) and transient.

Permanent faults were injected from the start of the binary execution, and the checkpoint analyzed was the last checkpoint after the execution end. The faulty structure was the intRF [1] and bitPosition and bitLine were pseudo-randomically chosen.

Transient faults are the same to configure despite the fact the injection point is pseudo-random after the binary is loaded into memory. Since the transient fault is just a flip in a random place in a structure, it is better to make random the injection time to have a better diversity in the data.

Faults results. As it is discussed before, each fault injection can lead to different results. It is then useful to understand which are the rates of the 3 categories to have an idea on how to collect sufficient data, Figure 2.4 2.5 2.6.

Binaries The injections were made on some MiBench [17] C programs:

- *qsort:* sorting algorithm
- *basicmath:* simple C math library testing (square root and angle conversion)
- *bitcount:* different algorithms to count number of bit equals to 1 in a given number
- *ssearch:* algorithm to find a string inside another string

[1]Register files are temporary storage locations inside the CPU that hold data and addresses.

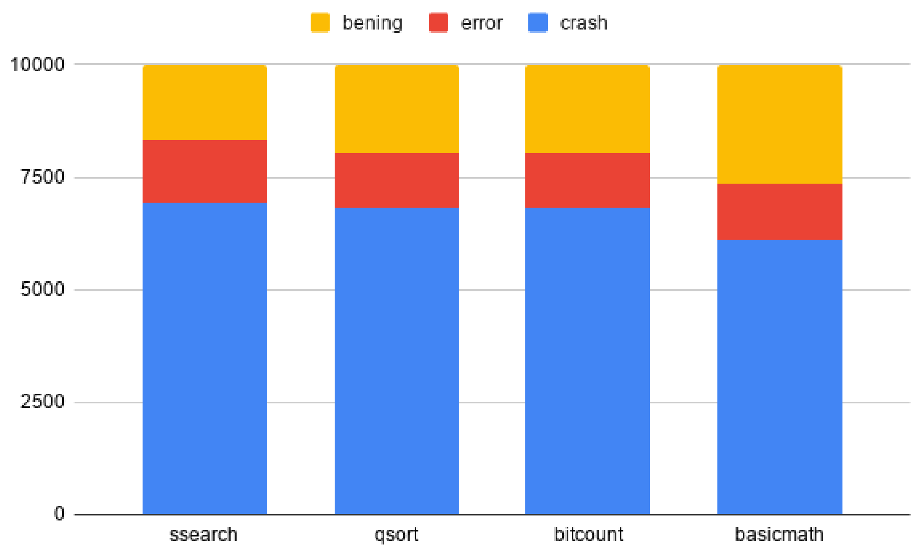


Figure 2.4. Faults results stuck-at-0

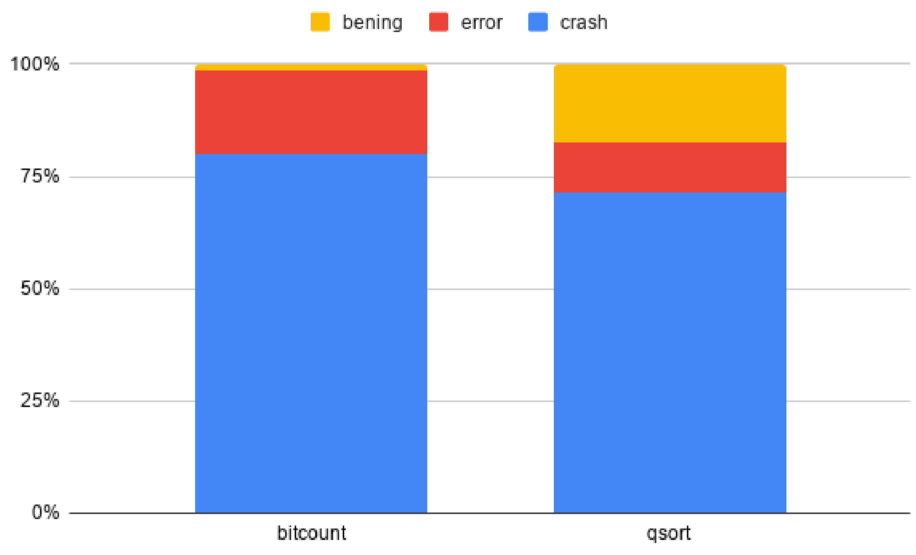


Figure 2.5. Faults results stuck-at-1

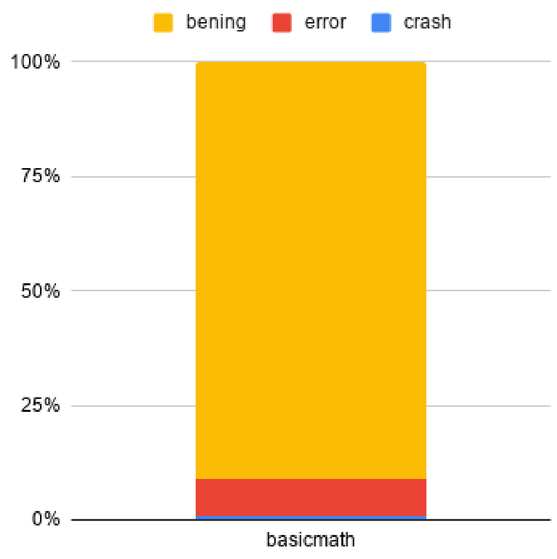


Figure 2.6. Faults results transient

Note. Crashes lead to discarding the simulation.
In addition it is possible to notice transients produce a lot of benign errors and stuck-at-1 tends to result in crashes instead. And this is aligned with the brief descriptions I gave in subsection 2.2.2.

2.3.2 Simulation Statistics

The statistics file is generated after the end of the simulation. It is compelling to figure out how this is generated because it is the center of my analysis. Each statistic is originated dynamically because each SimObject defines a set of statistics though Stat objects as attributes. The Stat interface defines methods to initialize, give a description, set value type (counter, average, percentage), etc. Then, values are updated after the `nextCycle()` function, or wherever we have access to them, and finally each Stat object of each simulated SimObject is dumped in a file with name, value, and description.

```
---------- Begin Simulation Statistics ----------
sim_seconds  0.047551  # Number of seconds simulated
sim_ticks  47550712000  # Number of ticks simulated
final_tick  5171294315000  # Number of ticks from beginning
sim_freq  1000000000000  # Frequency of simulated ticks
host_inst_rate  681632  # Simulator instruction rate (inst/s)
```

Listing 2.3. Example of statistics file

2.3.3 Simulations Hardware

gem5 and FIMSIM add-on were built inside a Docker running Ubuntu 14.04.6 LTS.
All of gem5 runs were executed on 32 Intel(R) Xeon(R) Silver 4110 CPU @2.10GHz, 93GB RAM and 5.5T disk space.
Each simulation lasts approximately 80 seconds, and it takes 6.4MB of disk space.
Here there is the bash script to launch several simulations and defining check-points and fault injections.

```bash
# $1 is the start number of simulation
# $2 final number of simulations,
# $3 defines how many simulation in parallel
HOMEDIR=/FIMSIM
TESTDIR=/simulatore
#define binary name
ap="bitcount"
prog="bit"
i="1"
#define tick to finish simulation
cp_final=5600000000000
#definition of fault parameters
echo "0 intRF -1 -1 30000 0 stuck-at-0 -1 1000" > ${HOMEDIR}/
    ffault.txt
for ((i=$1; i<=$2; i+=$3))
do
    all_pid=()
    for((j=0; j<$3; j++))
    do
        l=$((i + j))
        # setup folders to start from a checkpoint
        mkdir ${TESTDIR}/${ap}_${l}
        cp -rf ${HOMEDIR}/faultInjectionScripts/goldenCPSpec/${
    ap}/cpt.5123743603000 ${TESTDIR}/${ap}_${l}/.
        # launch simulation with a timeout to avoid
        # a simulation being stuck
        timeout 60s ${HOMEDIR}/build/X86_docker/gem5.opt -f ${
    HOMEDIR}/ffault.txt -s ${l} -d ${TESTDIR}/${ap}_${l} ${
    HOMEDIR}/configs/example/fs.py -n 1 --checkpoint-dir ${
    TESTDIR}/${ap}_${l} --take-checkpoints=${cp_final
    },50000000000 --max-checkpoint=1 --caches --script=${HOMEDIR
    }/run_${prog}.rcS -r 1 &
        # add pid to pidlist
        pid=$!
        all_pid+=( $pid )
    done
    # wait for processes to finish before starting another bash
    for p in $all_pid
    do
            wait $p
    done
    rm ${TESTDIR}/${ap}*/cpt*/system* #remove useless files
done
```

Listing 2.4. Bash script to launch simulations

Chapter 3

Data Analysis and Models

In this chapter it will be explained the data exploration, features selection and finally the models implemented. Alongside performances will be discussed, and contextualized with mathematical details and rationale behind each choise.

3.1 Data Engineering

Data engineering is a big part of a classification model because firstly it helps to understand how models can be constructed and secondly it is necessary to interpret results.
Usually, data preparation is split in: data extraction, data cleaning, data enrichment (this step will be skipped because there is the necessity in this case), and finally data are ready to be used.

3.1.1 Data Preparation

Several simulation statistics has been collected, they are different from the collection perspective (different binary or different fault type) but they are similar in the structure, so in the next paragraph there will be the general data preparation pipeline used for each different dataset.

Data Extraction After having a folder with all sub-folders regarding the simulations, it is necessary to extract the actual data to build the *cvs file*. First of all, the fault's results are decided with a script running the decision process I explained in section 2.2.2.

This bash script generates a file with name *{fault(error), nofault(benign error)}* or no file if there was a crash of the simulator. After that, subfolders containing the fault result files will be inspected by a Python script to extract statistics from the *stats.txt* file and add the fault label. The Python script avoids collecting percentage or statistics containing multiple values (30 of 600 in total) for simplicity reasons and because usually each statistic is replicated in different formats so dropping some values is not a big deal.
In this way is generated the dataset with one entry with all the attributes for each simulation that did not crash. It is very simple but it is important to add a data cleaning process because bash and python scripts don't handle missing values.

Data Cleaning Since the stats are generated dynamically at each run the number of features is not fixed for reasons that are not cited in the paper or explained in gem5 documentation. Therefore sometimes some statistics are bad formatted, absent, or present for the first time after 100 equal simulations. For this reason, I will refer to the number of features approximately and I put in place three steps cleaning process:

- Drop feature columns with more than 20 entries as Not-a-Number. Since there are a lot of features it is better to drop columns containing not well-collected data. This process approximately takes out up to 30 features.

- Drop entries with NaN values. This is to avoid handling missing values. Since I have enough entries to train models this process does not cause much loss of information.

- Drop semi-constant columns with less then 1% variance. This process takes out half of the atributes. This is not surprising because there are a lot of constant metrics, for example the voltage domain of the CPU, that will not change if there were a fault or not.

Data Domain Domain, as it will be clear later, is a big part of the analysis. Different datasets for different binaries were collected, these datasets are of course different but structurally the same, and to distinguish one from another it is usually said they are coming from different *domain*.
The process explained before is applied to all datasets, there will be domain distinctions when the model is trained on the actual data.

3.1.2 Features selection

At the end of the below process it remains \300 features and the fault label, if the fault detection mechanism is built inside a OS is safe to say for the majority of these statistics there is not a monitor for all of them already implemented. For this reason, it is good to evaluate a features selection that results in a minimum set to make models work reasonably good.
All of the following analysis have been made on a single dataset, that will be used to build the baseline for all the future models.

Pearson Correlation There are several ways or choosing a feature selection. The most common is to find which features are the most correlated to the label.
Pearson correlation expresses how linearly related are two features: the higher the absolute value the more related they are.

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} \tag{3.1}$$

Where σ is the standard deviation and *cov* is the covariance between X (feature column) and Y (the label column).
After having calculated the topmost correlated attributes to the label we choose them to be our feature selection.
This method is very simple yet it has some major problem, one of them is that the attributes selected can be highly correlated between them and so the selection cannot conserve much variance of the original data.

PCA Another method to make dimensionality reduction is *Principal Component Analysis*. Principal Component Analysis functions by transforming a large set of variables into a smaller one that still contains most of the information. In particular, each principal component is perpendicular to the preceding and it points toward the maximum variance and minimum reconstruction error. This graph is useful to comprehend how many principal components we need to express a certain amount of variance. It is visible that it is convinient to reduce the dimensionality of our datasets with PCA (99% variance explained with 19 components). However, even PCA for the dataset would have been very powerful it was not applied because it makes the attributes losing their metrics meaning, which will be fundamental to build the OS monitor later on.

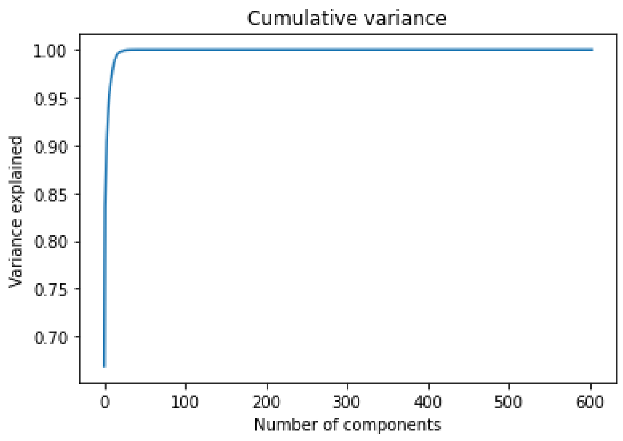


Figure 3.1. PCA cumulative variance explained

Feature Importance Another way to evaluate a feature selection is to build a model, like a Random Forest [7] and calculating the features importance factors. The important factor expresses how each feature decreases the weighted impurity in a tree, so how much each feature is important to build the best model.

$$\begin{aligned} fi_i &= \frac{\sum_{j: node\ j\ with\ split\ on\ feature\ i} ni_j}{\sum_{k \in allnodes} ni_k},\ ni\ is\ the\ gini\ importance\ coefficient \\ normfi_i &= \frac{fi_i}{\sum_{j \in allfeatures} fi_j},\ normalize\ with\ respect\ to\ other\ scores \\ RFfi_i &= \frac{\sum_{j \in alltrees} normfi_i}{Number\ of\ trees},\ average\ for\ all\ the\ trees \end{aligned} \tag{3.2}$$

Using a Random Forest helps to avoid searching only for linear relationships, as with PCA or correlation coefficients. In this figure 3.2 there are the features importance out of the 600 initial attributes, remember half of them are constant values.
It is also important to avoid having in between high correlated features selection, this is because highly correlated attributes will bring the same information to the model and sometimes having too many highly correlated attributes can even worsen the performances. In Figure 3.3 it is possible to see these correlations graphically.

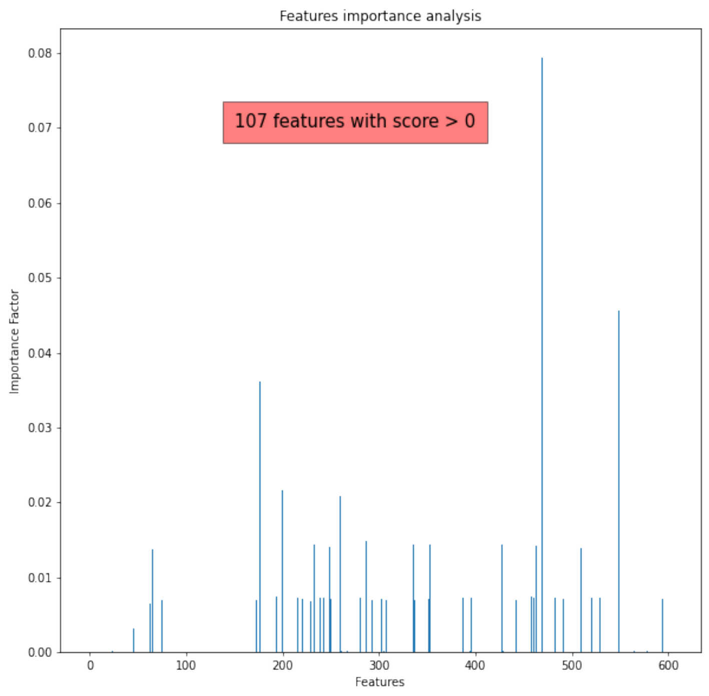


Figure 3.2. Feature Importance Score for the 600 attributes

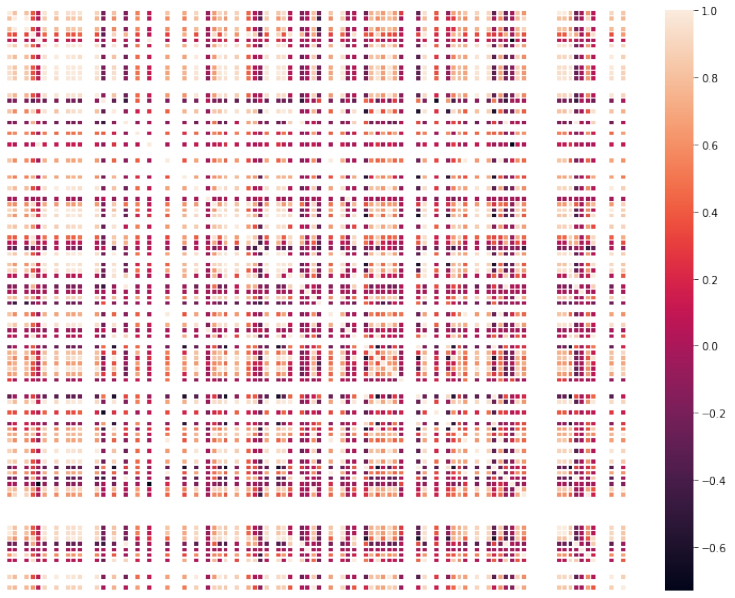


Figure 3.3. Correlation of 107 features with importance score > 0

Final Selection In a fashion of keeping the things as simple as possible, the top 20 importance score features is selected to be the final set. In this way, the number of features is as low as possible and the meaning of each feature is save.
Alongside this selection, I will consider the totality (300) of features with types of models requiring a high number of features (ex. CNN).

3.1.3 Training and Test Procedure

For the training procedure the classical three-way split has been adopted:

- *Train set(50%)* is used to train the model,
- *Validation set(20%)* is used to tune the model. Validation is used to avoid 'cheating' and tune the model to have the best performances just in the test set and not in general.
- *Test set(30%)* is the ultimate test to evaluate the performances of models on unseen data

The important concept to understand is the *seen/unseen difference.* The *seen* is all has to be known during the training: for example the train set is known, its attribute distribution, its features importance. The *unseen* is what the model cannot know before test time: an example is the feature importance of test or validation set, either for example if the test set is normalized it needs to be scaled with the mean and variance of the train set. So, in every choice is important to keep in mind the model is trained on the *seen* and tested on the *unseen*, in this way we are sure if the model performs in the unseen is because the model was able to generalize.

Domain and Testing Domain is a part of the analysis, so it is introduced the concept of domain testing. Basically, I have different datasets and each one can be trained and tested separately but obtaining the best performance on a single dataset is not the main goal of this work. The idea of this thesis would be to obtain a general model, able to work on different binaries without major training phases.
So, two different testing procedure are used:

- For transfer learning models: train/validate on a domain, train/test again on another domain.

- Cross-domain validation for domain adaptation models: in this case, the model is trained on two domains (source and target), then it is validated on other domains to avoid 'cheating' and test on target domain.

3.1.4 Feature Analysis

In this section I will list the top features I have used for my dimensionality reduction procedure.
The features are the top 19 most correlated features to the fault label in the basicmath binary dataset. The description is given referring to configuration created in section 2.1.3. I report a figure to make the descriptions clearer.

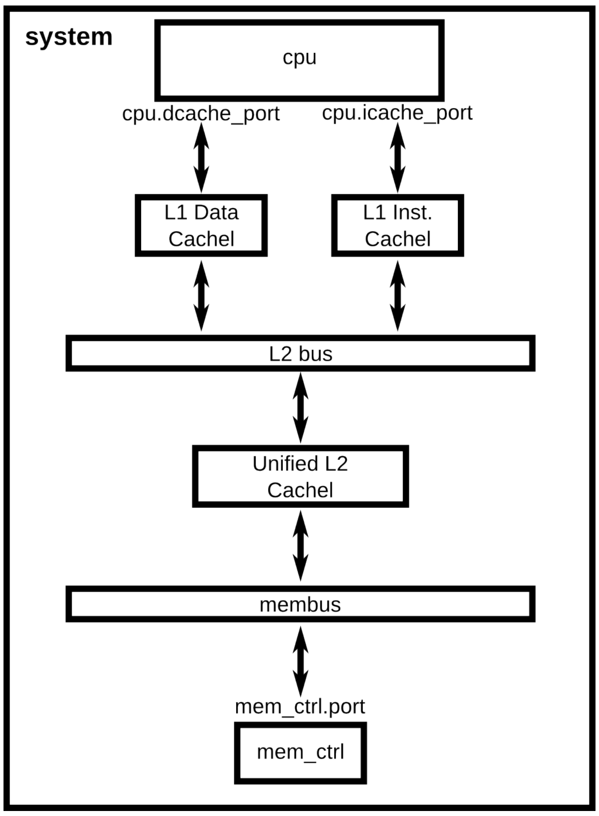


Figure 3.4. Architecture Hardware Configuration. Figure from [5].

Instruction TLB The Translation Look-aside Buffer is used as a cache to save recent physical memory to user memory translation. It is faster and smaller than a cache, and it is used in the instruction pipeline. A TLB is

composed by a value and a tag, the tag is used to retrieve the value.
The first is the average number of references to valid (the value is up-to-date) block, the next three are counter related to the number of Read Request accesses and hits(when the value searched is present), the fifth is the average of occupied block in the tbl and the last is the overall number of misses.

```
Statistics name: system.cpu.itb_walker_cache.tags.avg_refs,
system.cpu.itb_walker_cache.ReadReq_accesses::cpu.itb.walker,
system.cpu.itb_walker_cache.ReadReq_hits::total,
system.cpu.itb_walker_cache.ReadReq_accesses::total,
system.cpu.itb_walker_cache.tags.occ_blocks::cpu.itb.walker,
system.cpu.itb_walker_cache.overall_misses::total
```

Data TLB It is the same concept of the metrics before. The first counter in the number of demand (read+write request) accesses and the second is the number of misses.

```
system.cpu.dtb_walker_cache.demand_accesses::total,
system.cpu.dtb_walker_cache.demand_misses::cpu.dtb.walker
```

Instruction Cache Instruction is the level-1 cache used to store instructions. The first counter represents the time passed by a task in the cache by task id 4, the second is the number of demand access, the third and the fourth are the number of misses for CPU instruction and hits for read request, and the fifth are counters related to the number of hits in the cpu for the cpu instructions.

```
Statistics name: system.cpu.icache.tags.age_task_id_blocks_1024::4,
system.cpu.icache.demand_accesses::cpu.inst,
system.cpu.icache.ReadReq_misses::cpu.inst,
system.cpu.icache.ReadReq_hits::total,
system.cpu.icache.overall_hits::cpu.inst,
```

Data Cache Data Cache is the level-1 cache used to store data, and this counter is the number of misses cause by SoftPF requests, which are a particular type of read request.

```
Statistics name: system.cpu.dcache.SoftPFReq_misses::total
```

IO Cache IO cache is the cache for IO requests. This counter is referred to the number of hits for write invalidate requests(WriteInvalidateReq ensures that a whole-line write does not incur the cost of first doing a read exclusive, only to later overwrite the data).

`Statistics name: system.iocache.WriteInvalidateReq_hits::total`

Memory Bus These metrics refer to the bus linking cache to memory controller. The first refers to the total cumulative size of every packet flowing into the membus from the instruction cache to the memory controller and the second is referred to cumulitive number of packets flowing from the TLB to the memory controller.

`Statistics name: system.membus.pkt_count_system.cpu.dtb_walker_cache.mem_si`
`system.membus.pkt_size_system.cpu.icache.mem_side::system.mem_ctrls.port`

IO bus The IO bus carries the IO request from devices to the CPU. The transation distribution is counting the different type of commands, this metric measures the number of read requests flowing in this bus.

`Statistics name: system.iobus.trans_dist::ReadReq`

Memory Controller The memory controller is a circuit in charge of managing the flow of data from the CPU to the main memory.
This counter is the number of read request for CPU instruction.

`Statistics name: system.mem_ctrls.num_reads::cpu.inst`

Final consideration Some of the attributes I have presented sometimes are really similar or even equals and this can be redundant or useless. However, I have decided to keep all these 19 attributes because the dimensionality reduction was already enough (600 $\rightarrow$ 19) and in some more complex simulated hardware architectures things could be change slightly and these attributes could be needed.

3.2 Models Analysis

Faults detection offers multiple benefits, improve security, reliability and most importantly can improve performances of a system. In the past, faults detection or health analysis was made through physic-based model [49]. Because the problem faced in this work involves too much components to be modelled different ML models will be evaluated to solve faults detection.

3.2.1 Introduction to machine learning

Arthum Samuel invented the word *machine learning* in 1959. The main aspect of these algorithms is the ability to build a mathematical structure over the so-called train-data and then being able to make predictions without being programmed to do so.
Later machine learning has become a huge word in computer science and it is used pratically everywhere. However, before starting to enter in more complicated models a simple Feed-forward Neural Network will be presented. It is one of the simplest machine learning models and having solid knowledge on this will be useful later on.
Feed-Forward Neural Network [39] is the simplest deep neural network existing, usually used for classification. Each layer is composed of several neurons and the network is composed of several layers. Each neuron is based on a

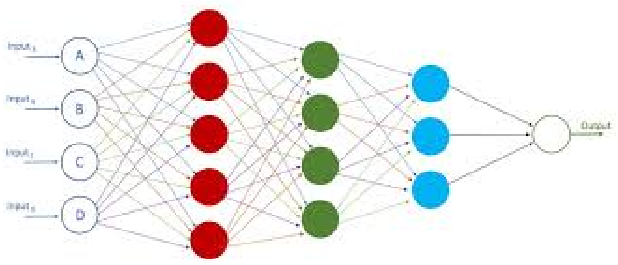


Figure 3.5. Feed-forward Neural Network. Figure from [50].

simple linear algebra operation:

$$y = \sigma(\sum_{i=0}^{i} w_i^T xi) \tag{3.3}$$

Y will be the response variable (0 or 1 in a binary classification), x will be the features vector and w is the unknown weights vector.
The σ function is called activation function and it is used to map values and add non-linearity. The simpliest example is the sigmoid, which squeezes the values in the range [1,-1]
Now the question is: how do we find w?

Learning Algorithm. The simplest learning algorithm is mini-batch learning and goes like this:

- Calculate the loss through a loss function: this function expresses how different is the result from the expected value.
- The loss is backpropagated into the layers, each neuron will receive a fraction of the total signal of the loss dependently on how much it contributed.
- This portion of the loss is then concordantly used to update the value contained by each neuron (the w).

This is repeated for each data batch in the train set. The idea behind this learning process is to minimize the loss (total loss equals to 0 $\rightarrow$ data points classified correctly).
To update the weight it can be used the so-call *gradient descent.* Gradient descent uses the first derivative (gradient) of the loss function when updating the parameters. The process consists of chaining the derivatives of the loss of each hidden layer from the derivatives of the loss of its upper layer, incorporating its activation function in the calculation. The weight in each neuron is updated in the opposite direction of the gradient, so the loss tends to a minimum.

$$\begin{aligned} \Delta w &= \nabla w L(x_i, y_i, w), \ \ \textit{For each neuron in each layer} \\ w &= w - \lambda \Delta w, \ \ \textit{Where } \lambda \textit{ is the learning rate} \end{aligned} \tag{3.4}$$

This is repeated for each neuron for each layer for each batch. In this way, the network *learns* in a sense that tends to a state where the loss on the train set is very close to zero, so if the train set is well-built the network will be general enough to be used for classification.

Limitations. Of course this very simple model has a lot of limitations.
The first one is that it is very slow to converge, the second is that with many neurons the learning phase is very slow and computational expensive and the third is that for the preceding reasons it can be really difficult to obtain a general model without any tweaks.

Improvements. To overcome some of the limits computer scientists came up with different solutions:

- *Different architectures* like CNN [13] to fasten train phase and to generalize better or RNN [41] to evaluate time as a variable of the model.
- *Different learning algorithm* like Adam or SGD.
- *Different layer construction* like batch normalization layer or different activation function

3.2.2 Fault detection: related works

Fault detection has been an interesting field of studies in the last year. The dataset used is custom-made and these works that will be presented here come from a wide range of different domains(IoT device, Cloud-based data centers, HCP systems, bearing systems, semiconductor materials), so the concept of each model structure will be explained to give an idea where the models that will be utilized later took ispiration from.

FFNN and Transfer Learning FFNNs, explained in the section before, are very powerful and they can model non-linearity very well, however they tend not to work well with domain-shift. Hence, researchers invented a new learning process. At first, the network is trained fully on a dataset, secondly, we freeze the some layers and just train the remaining layers on another dataset(usually smaller and in less epochs). This makes sense because the different layers carry different information, like in object-recognition the first layers 'search' for shapes and shapes recognition is good for each domain so the first layers do not have to be trained each time [52].
In conclusion, it is possible to create different specialized models for each domain.
In this work [53] they focus on how difficult is to obtain enough data for different rotary equipment to build a reliable fault detection model. So, they used simulations to build the general model and then train specifically for different components with less data and time.

Deep Belief Network Deep Belief Network are a special type of Deep Neural Network model, that can be seen as a stack of multiple restricted Boltzmann machines.
RBM is composed of 2 layers: one visible and one hidden, and it is used to learn a distribution. The *train procedure* consists of gradient-based contrastive divergence algorithm and backpropagation. Basically, in the first

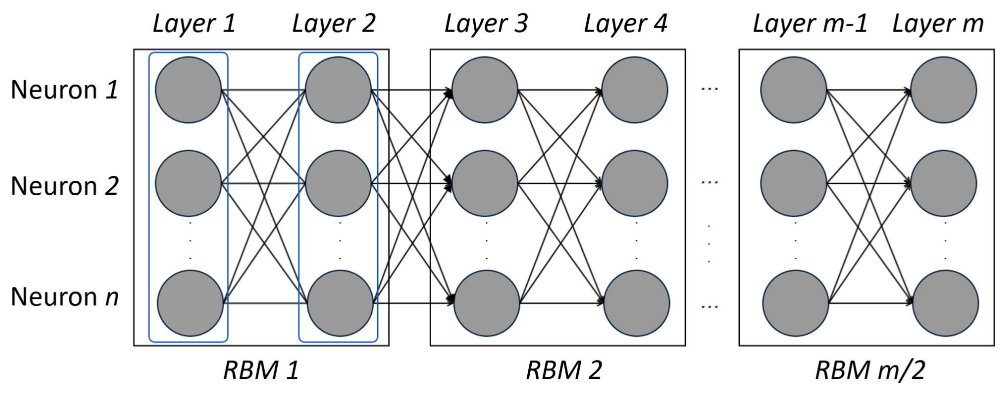


Figure 3.6. Deep Belief Network. Figure from [35]

phase weights are updated for each sample transforming from visible to invisible layer (positive phase), then the input is reconstructed passing from hidden to visible (negative phase).

$$\Delta w = \lambda(\langle v_k, h_k\rangle_{data} - \langle v_k, h_k\rangle_{recon}) \tag{3.5}$$

Where λ is the learning rate, v and h are the states of the layer after the positive and negative phase.
Intuitively, the RBM tries to learn the distribution by premiating how much 'information' of the input is carried by the output. What happens with DBN is that each RBM is trained one after another.
After that, the model is trained as an FFNN with backpropagation and labeled data.
DBN is used from Tamilselvan [45] to analyze the health state of an engine through sensors data, reaching very good results because the strength of DBN is tending to converge faster and better because the backpropagation is used just to fine-tune the model after it has been initialized by RBMs unsupervised training.

Fuzzy Neural Network Fuzzy Neural Network is an enhancement of FeedForward Neural Network to handle instability and uncertainty in the data. Each feature is converted into a fuzzy set and to each category in the set is assign a degree of membership. This assignment is embedded into fuzzification layer, and it precedes a FFNN classifier.
Steady, *Increase*, *Decrease* is defined in relation of a single IoT measurement with respect to the average of the others. This technique was used in an IoT water monitoring system to detect faults of one of the IoT systems [10] and

Input	Fault Symptom	Decrease	Steady	Increase
X1	Water Temperature	0.1	0.5	0.9
X2	Supply Voltage	0.2	0.5	0.8

Table 3.1. Fuzzy Set definition

it obtained very good results. The main problem is that the choice of membership degree relies on experts, and it needs multiple sensors measuring the same metric.

CNN FFNN bears with a problem: as we see in Figure 3.5 layers are fully connected (each neuron is connected with each neuron of the following layer), so the number of operations for each train step is very high.
Hence, convolutional filters were put in practise: essentially a convolution filter is used to reduce the dimension of data without loosing too much information, expecially if close features are related (like pixels in a picture).

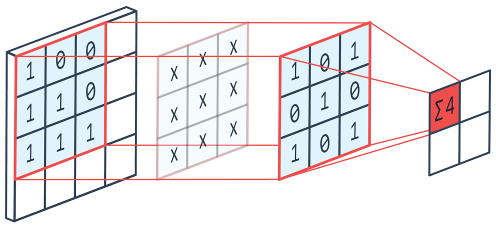


Figure 3.7. Convolutional filter. Figure from [8].

A CNN is composed of several convolutional filters and an FFNN to transform the features extracted by filters in the final classification result.
CNNs are also used for fault detection in bearing systems. In this work [24] the input is vibration data referring to bear fault. Lately, CNNs were also applied to semiconductor manufacturing processes by Lee et al. [25] in a very clever way. Sensor data are aggregated to be used as a matrix: each row is a set of measurements in a certain time. In this way also the time is a 'feature' without recurring to more complex architecture like RNN.

AutoEncoder Another approach is using unsupervised models: the difference is that with unsupervised models data does not have to be labeled.

AutoEncoders, Figure 3.13, are very similar to RBM, but the structure and training are different. Basically with RBM stochastic representation of the data is searched and the training phase is done from the two sides of the RMB. While AE is composed of an encoder (mapping the input to a lower dimension), and a decoder (reconstructing the output of the layer before back to its dimension), and the loss is the difference between the input and the reconstructed input (also called reconstruction error).
It is possible to have multiple encoder layers followed by multiple decoders, this model is called Stacked AutoEncoder.
Bolegnesi et al. [6] used metrics and data collected by monitors installed on

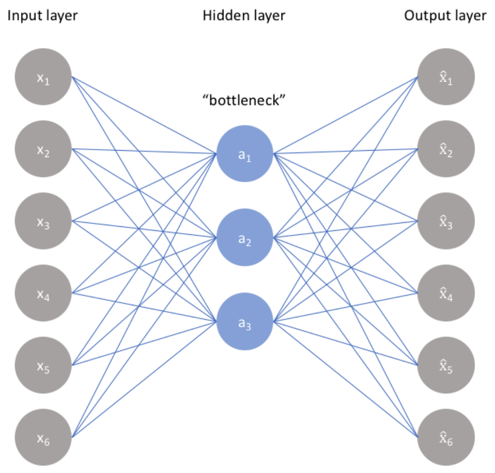


Figure 3.8. AutoEncoder. Figure from [20].

HPC. They trained AEs to work with non-labeled data coming from an HCP. After having calculated the mean reconstruction error for the training phase, they stated that if the reconstruction error of a run in a node is particularly higher relative to the training phase's one then there has been a fault. As a fault, they considered hardware faults, misconfigurations, software bugs, basically any 'problem'. This approach is a very handful because it does not need any faulty run sample to work.
Another way of using AEs is to remove the decoder and use the encoder as a features compression network. The concept behind that is that the decoder should learn the latent representation of the data. In this paper [31], the authors trained an S-AE on data corresponding to patients with no heart disease, and then they used the decoder followed by a FFNN classifier. This improves the performance of the models because the SAE before the classifier

helps to 'distantiate' positive cases from negative cases.

Spiking Neural Network Non so se metterlo ma lo lascio un attimo qui

3.2.3 Proposals

All of the modelspresented before are related to different datasets, for the dataset analyzed there is a domain issue to solve. Different faults on different binaries produce different outcomes and one model cannot cope with differences.
In the next two paragraphs 3 models will be presented, they are different in the concept and in the implementation and it is a attempt to see what works better on this type of problem.

FFNN-Transfer Learning This model is very simple yet powerful, it makes use of the 19 features selection analyzed in section 3.1.4, and it is inspired by the work [53].
The rationale is straightforward: train the model on a binary dataset for N epochs and then specialize the model to work with another dataset training it for $N/5$ epochs.
The model, Figure 3.10, is composed of several components:

- *Linear layer:* it applies a linear transformation to the data.

$$y = \sum_{i=0}^{i} w_i^T xi$$

- *Batch Normalization Layer*: models benefit from normalized data because it squeezes data in a smaller range and it helps avoid a feature that prevails on others and big weight updates which cause 'bouncing' around the minimum.
  Usually, data are normalized before entering the model but then it is not predictable what will happens after the first layer. So, batch normalization layer comes in place and normalizes output after the linear transformation.

$$y_{norm} = \frac{y - \mu_{batch}}{\sigma_{batch}}$$

  In addition, batch normalization helps the train phase by permitting higher learning rates, by reducing the effect of bad initialization and by containing the so-called covariate-shift effect [19] (the higher the effect the lower is the generalization).

- *Dropout Layer*, Figure 3.11: this layer is in charge of excluding a neuron from the net with a certain probability (usually 50%) at each train-step. Dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs, and this reduces overfitting. [43]

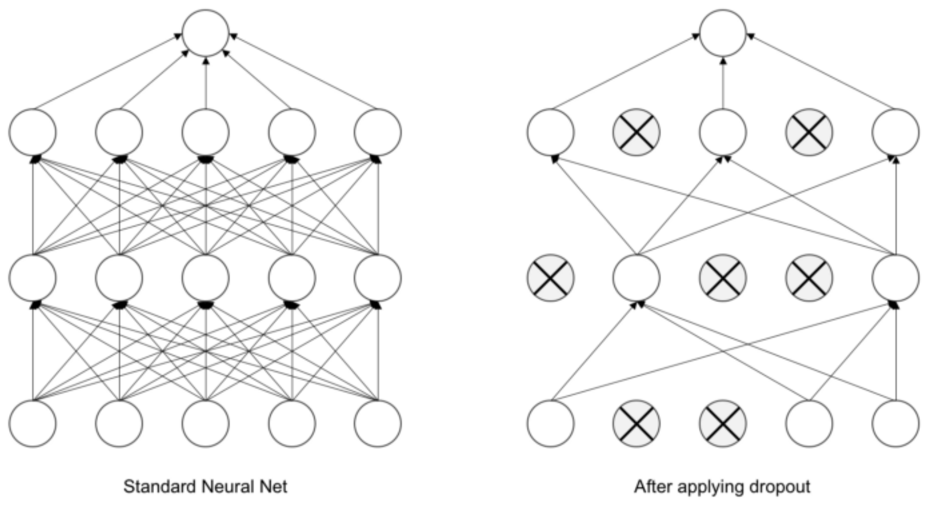


Figure 3.9. Dropout functioning. Figure from [40]

- *Leaky Relu Layer*: this is the activation function that maps the output of the linear transformation into a number in a certain range to be passed to the next layer.

$$f(x) = \begin{cases} 0.01x, & \text{if } \; x < 0 \\ x, & \text{if } x >= 0 \end{cases}$$

It is the most used activation function because it helps train faster because it does not saturate, unlike sigmoid, and prevent neuron death in the negative region. region.[29]

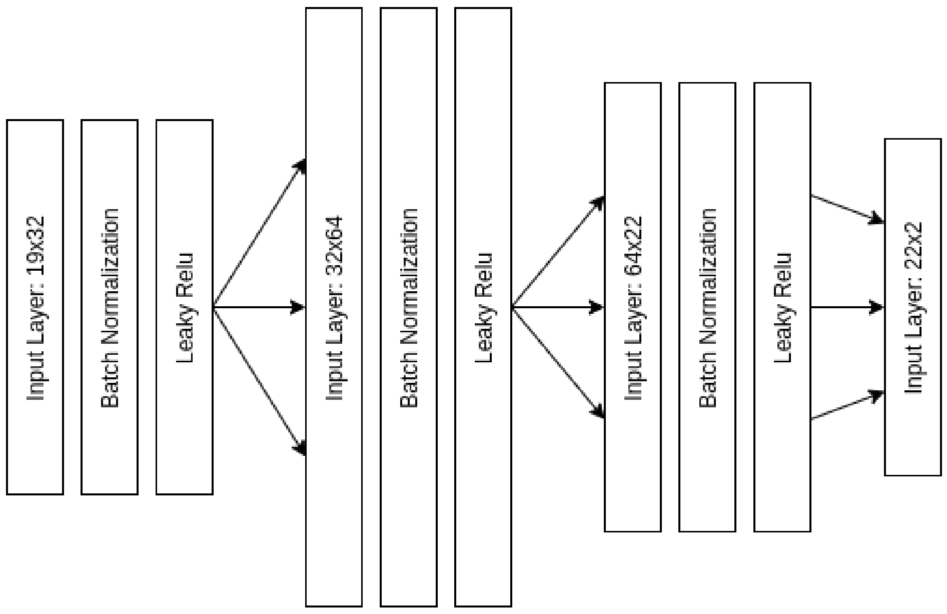


Figure 3.10. Neural Network

The model is trained using:

- *CrossEntropyLoss*: the loss function defines how the model learns. Cross-entropy loss is the standard for categorical problem but it can be used for binary classification too.
  It is calculated:

  $$loss(x, class) = -\frac{\exp(x[class])}{\sum_j \exp(x[j])}$$

  Where *class* is the correct class of the x, and x is the vector containing the score for each class(N=2). This formula it is iterated and averaged for all points in a minibatch.
  The final score summarizes the average difference between the actual and predicted probability distributions.

- *Adam optimizer*: vanilla Stochastic Gradient Descent is good but tends to stuck in local minima or saddle point. So researchers found it was a good idea to add momentum to build up velocity to overcome saddle point or saddle.

  $$m_t = \beta_1 m_{t-1} + \nabla L(x_{t-1})$$

  $$w_t = w_t - \alpha m_t$$

β_1 is the friction to avoid velocity to build too quickly (usually 0.9), ρv is the momentum, α is the learning rate and L is the loss function. Just with momentum the step starts to get very big very quickly, so these steps can be regularized by adding another term.

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\nabla^2 L(x_{t-1})$$

Then the estimators are corrected (bias correction), for the fact that first and second moment estimates start at zero.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

At this point, the final weight update will be:

$$w_t = w_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$$

Adam [22] optimizer helps overcoming area with tiny gradient while avoiding big updates, in addition the updates are bounded to α which is the only parameters to tune.

- *StepLR scheduler*: it performs the very simple operation to scale the learning rate after a number of epochs by a factor.

  $$\alpha_{new} = \gamma \alpha_{old} \text{ After each N epochs}$$

  The parameter to choose are N and γ. This allows large weight updates in the beginning of the learning process and fine-tuning towards the end of the learning process.

- *CycleLR*: this is based on another way of scheduling the learning rates. The idea is very simple: the learning rate is scheduled in a triangular shape over the epochs. The 3 parameters to define are: initial lr, final lr, step size.

  According to this paper [42] this scheduling technique with SGD+momentum permits to obtain very good performances in few cycles.

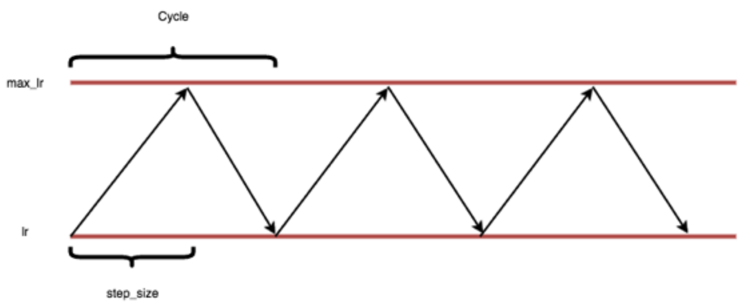


Figure 3.11. LR cyclical learning. Figure from [21]

DANN Domain Adversarial Neural Network [15] is a model to cope with domain adaptation problems, where data at train and test time are structurally similar but distributed differently.
The approach consists of training on a so-called *source domain* labeled data and *target domain* unlabelled data. The resulting model should then be able to perform on the task independently from the domain.
Since the description before seems to fit with the datasets collected for this work, I decided to use DANN. Intuitively metrics are like pixels, close metrics are near because they refer to the same hardware for the way metric output is organized. For this reason, I thought it could be a good idea to use DANN with 1d Convolution Layer on the totality of features .
The model, Figure 3.15, is composed by:

- *1d Convolutional Layer*: it is in charge of performing the convolution operation on the input.

$$(N, C_{in}, L) \rightarrow (N, C_{out}, L_{out})$$

$$Out(N_i, C_{out}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \star input(N_i, k)$$

  C_{in}/C_{out} refers to the number of channels in input and output, N is the batch size, $\star$ is the cross-correlation operator which is the main difference with a Linear Layer, k is the size of the filter($weight$), L is the length of the signal.

- *Relu Layer*: this layer is used to define the activation function. The difference with Leaky ReLU is that ReLU is 0, for x<0, and it is the de facto standard for CNN.

- *MaxPool Layer/AvgPool Layer*: the pooling operation is used to downsample the features maps obtained by the convolutional layer. It is used to regularize the model and avoid filters being too sensible to minor movements in the input.

$$(N, C, L) \to (N, C, L_{out})$$

$$Out(N_i, C_{out}) = \underset{m=0,..,kernel\_size-1}{oper} input(N_i, C_j, stride \times k + m)$$

The operation applied to each patch define the layer: max is a MaxPool Layer, *Avg* AvgPool Layer.

- *Gradient Reversal Layer*: this layer places a simple operation. Its use will be clear later.

$$Grad_{new} = -\alpha \cdot Grad_{old}$$

- *Task Classifier*: FFNN similar to the one explained in paragraph 3.2.3, it is used to classify the fault label.

- *Domain Classifier*: FFNN similar to the one explained in paragraph 3.2.3, it is used to classify the domain label.

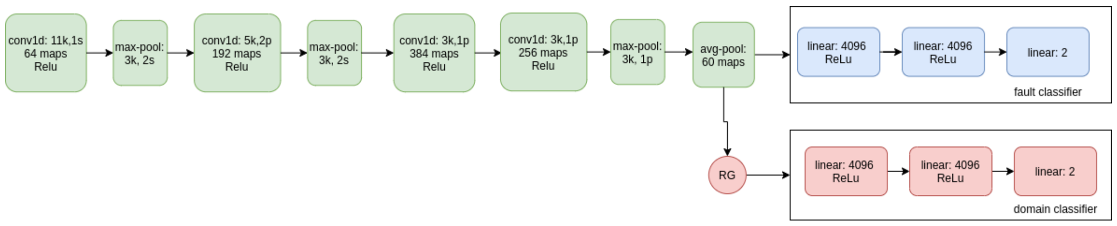


Figure 3.12. DANN: k=kernel of the filter, s=stride of the filter, p=padding of the filter, maps=output maps

The rationale is to have two branches after the features extractor: the label classifier is in charge of learning the actual task of the network, in this case the fault detection; the domain classifier learns to distinguish between the source domain and the target domain.
So, while the model in the first branch is learning the task, on the other branch the reversal gradient layer pushes the feature extractor not to distinguish between the domain while the domain classifier becomes better to guess it. The loss of the domain classifier will grow because the gradient used

to minimize the loss is reversed, so the model is updating toward domain invariance. And this is the reason why this model is called adversarial, because while the domain tries to learn to guess the domain the features extractor will make it progressively more difficult.
As a consequence, without target labeled data it will achieved a model able to learn the task independently from the domain, source or target. To train this model I made use of:

- *CrossEntropy Loss* for both domain and task classifier.
- *Adam Optimizer*: this is used to optimize both of the loss.
- *StepLR Scheduler*: this is used on both optimizer too.

Sparse Stacked Autoencoder+Classifier SS-AE is one of the main approaches to fault detection. It is used to 'transform' data before going into the classifier. I used it on the totality of attributes as a features extractor for a FFNN classifier.
The model, Figure 3.13, is composed of:

- *1d Convolutional Layer*
- *Relu Layer*
- *MaxPool Layer*
- *Task Classifier*
- *Upsampling1d Layer*: the upsampling layer is the opposite of the maxpool layer. In fact, it is used to upscale the input data in a very simple way:

$$X_{out} = \lfloor X_{in} \times \text{scale factor} \rfloor$$

$$[1\ 2\ 3] \xrightarrow{scalefactor=2} [1\ 1\ 2\ 2\ 3\ 3]$$

The concept here is to train the autoencoder (encoder+decoder) to reconstruct the input correctly, then we drop the decoder and we use the encoder as a features extractor for a fault classifier.
To do so the training phase adopts:

- *Loss*: the loss function is composed of different factors. Firstly a function to calculate the mean squared error between points.

$$MSE(X_{in}, X_{out}) = \frac{\sum_i^n (x_{in}[i] - x_{out}[i])^2}{n}$$

To improve the performances usually, it is added a term called L1 sparsity.
L1 sparsity forces the model to tend to lower weights values and this helps the model generalizing and not straight copying the input.

$$L1 = \lambda \sum_i |w_i|$$

The final loss function will be:

$$Loss = MSE + L1$$

- *CrossEntropy Loss*: this is used then to train the encoder+classifier.
- *Adam Optimizer*: it is used to train both autoencoder and classifier.
- *StepLR Optimizer*

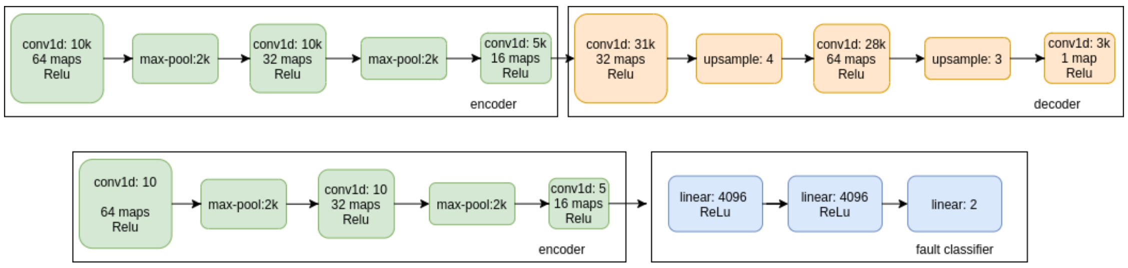


Figure 3.13. Stacked-AutoEncoders and Encoder+ Classifier: k=kernel of the filter, s=stride of the filter, p=padding of the filter, maps=output maps

3.3 Experiments

In the following section, the results obtained by the models will be discussed. However, there are many ways of evaluating a model. There is:

- *Accuracy:* is the simplest, and it is the number of correctly predicted points over the total.

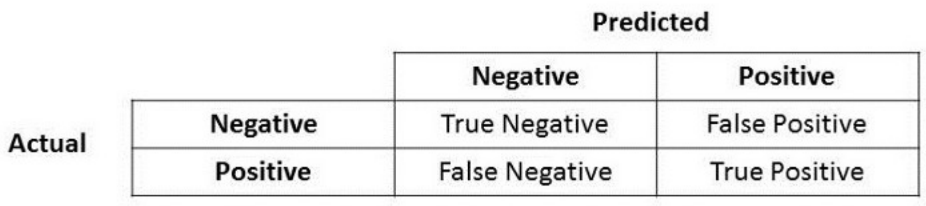
$$\text{Accuracy} = \frac{\text{Correctly Predicted}}{\text{Total}}$$

- *Precision and Recall:* accuracy is not able to cope with unbalanced classes and for some analysis can be misleading, like fault detection. In fault detection, it is interesting to find the faulty run much more than finding the non-faulty run. So it is better to use other metrics:

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive+False Positive}}$$

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive+False Negative}}$$

Here is the confusion matrix 3.14 to help navigate these definitions. So,



		Predicted	
		Negative	**Positive**
Actual	**Negative**	True Negative	False Positive
	Positive	False Negative	True Positive

Figure 3.14. Confusion Matrix

precision expresses indeed how precise we are when assign the fault label, recall instead gives a measure of how many faults we manage to find.

- *F1 score:* this score is defined to be a balance between recall and precision.

$$\text{F1} = 2\frac{\text{Precision} \cdot \text{Recall}}{\text{Precision+Recall}}$$

What is the best is not possible to choose apriori, the 'best' model is related to the application. Nonetheless, the f1 score highlights performances on fault class and the best models will be chosen according to it.

3.3.1 Neural Network and Transfer Learning

This section will make clear the validation process to build the baseline model (trained on one binary only) to be used the for transfer learning. Then, it will be shown the enhancement introduced by transfer learning.

Hyperparameters Tuning Evaluating a model it is essential to babysit the learning process, to make sure the network is learning well.
When a model is learning correctly the loss on train and validation set converge to the same values with the shape of an elbow. This specific shape expresses the loss of the network is decreasing to a point of stability and the gap between the train set and validation set means how much the train set is representative of the validation set.

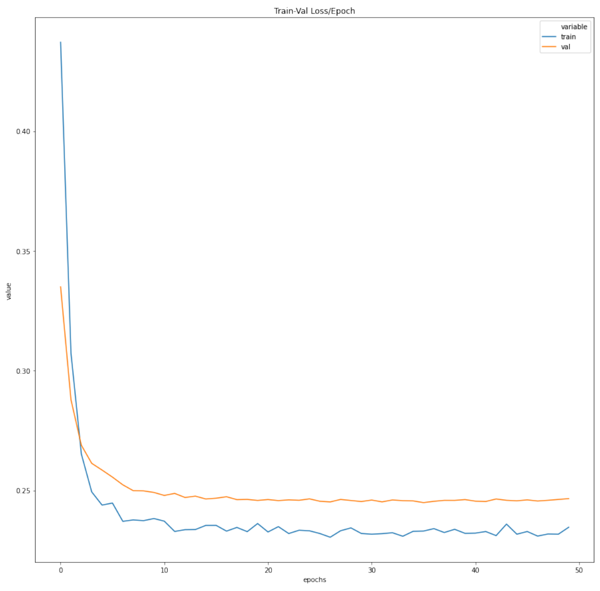


Figure 3.15. Loss over epochs, with LR=0.001 and 50 epochs

After ensuring the model is set up to learn correctly it is time to fine-tune it. As explained in section 3.2.3 the updates are bounded to the learning rate, the other two hyperparameters β_1 and β_2 are usually not changed and they are 0.9 and 0,999, both in Adam and SGD+Momentum. The scheduler and its parameters must be choosen as well.

Since all the performances are comparable, Table 3.2, I decided to keep the model with Adam, StepLR and the lowest LR.

Optimizer	Scheduler	Epochs	F1 score
Adam: LR=0.1, Beta1=0.9, Beta2=0.999	*StepLR: N=30 Epochs, gamma=0,1*	50	0.8693
Adam: LR=0.01, Beta1=0.9, Beta2=0.999	*StepLR: N=30 Epochs, gamma=0,1*	50	0.8693
Adam: LR=0.001, Beta1=0.9, Beta2=0.999	*StepLR: N=30 Epochs, gamma=0,1*	50	0.8693
Adam: LR=0.0001, Beta1=0.9, Beta2=0.999	*StepLR: N=30 Epochs, gamma=0,1*	50	0.8655
SGD+Momentum: LR=0.1, Beta1=0.9	*CycleLR: LR_f=LR, LR_i=LR*0.1*	50	0.8692
SGD+Momentum: LR=0.01, Beta1=0.9	*CycleLR: LR_f=LR, LR_i=LR*0.1*	50	0.8651
SGD+Momentum: LR=0.001, Beta1=0.9	*CycleLR: LR_f=LR, LR_i=LR*0.1*	50	0.8693
SGD+Momentum: LR=0.0001, Beta1=0.9	*CycleLR: LR_f=LR, LR_i=LR*0.1*	50	0.8195

Table 3.2. Validation scores to select the best baseline model

Results Now the best model with the last 2 layers froze is used as a baseline to perform the transfer learning task, remembering the 'baseline' has been trained on the *basicmath* binary only.
Table 3.3 is telling the model can work with different binaries, sometimes even maximizing the precision, with few epochs of training after having built the baseline.

Binary	Precision: baseline	Precision: after	Recall: baseline	Recall: after
basicmath	1.00	/	0.74	/
qsort	0.38	1.00 (+0.62)	1.00	0.82(-0.18)
search	0.46	0.82 (+0.36)	1.00	0.33(-0.67)
bitcount	1.00	1.00(+0.0)	0.73	0.74(+0.01)

Table 3.3. Performances on test set after 10 epochs of transfer learing

Disclaimer: An high recall is not always a good sign, if the model predicts always fault it will have 1 recall but it is not a good model. High precision indeed is a good measure of how the model is able to spot the faulty runs.

3.3.2 DANN

This model is trained based on a totally different approach. The concept here is to have a model trained on two different domains, one labeled and the other not, and see how the model performs on both domains after the training.
The validation set will be on the other two domains.

Hyperparameters Tuning The parameters to tune are related to Adam optimizer, so the learning rate, and the α, which represents how much reverse gradient is flowing into the features extractor.

LR/ALPHA	**0.1**	**0.01**	**0.001**
0.01	0.59	0.75	0.59
0.001	0.59	0.62	0.59
0.0001	0.59	0.76	0.75

Table 3.4. F1 score on validation set

Remember the model is not trained on domains coming from the validation set.

Results Having selected the best hyperparameters it is time to use this model using as target all the domains and see how it performs.

Binary	Precision: before	Precision: after	Recall: before	Recall: after
basicmath	1	\	0.80	\
qsort	0.23	1	1(+0.0)	0.77
search	0.36	0.46(+0.10)	0.8	1
bitcount	0.33	0.38(+0.05)	0.8	1

As it is noticeable, from unlabeled data the model is not able to generalize.

3.3.3 Sparse-Stacked-AutoEncoder

Hyperparameters Tuning Firstly it is trained the autoencoder, using MSE and L1 sparsity using only non-faulty runs.
The two hyperparameters to tune are the learning rate and the α L1 sparsity coefficient.
The encoder from the best model from above is then trained on the whole

LR/α	**0.01**	**0.001**	**0.00001**
0.01	0.0704	0.0427	0.0704
0.001	0.0355	0.0704	0.0462
0.0001	0.0355	0.0413	0.0704

Table 3.5. Best (the lowest) loss on non-faulty dataset on a single domain

(faulty and non-faulty) dataset on a single domain.
This model will be used as a baseline to confront performances on other domains before and after some epochs of training on them.

Results It is interesting to notice in the first column of Table 3.6 that this model has the best precision on unseen domains.

Binary	Precision: before	Precision: after	Recall: before	Recall: after
basicmath	1.00	\	0.80	\
qsort	1.00	1.00(+0.0)	0.20	0.78
search	0.86	0.83(-0.03)	0.11	0.32
bitcount	0.12	1.00(+0.88)	0.23	0.77

Table 3.6. Scores on test set before and after 10 epochs of training on the binary

3.4 Fault types

All the models and results discussed in this chapter are related to *stuck-at-0* fault datasets.
However, datasets regarding other type of faults 2.2.2 has been collected to make minor experiments in order to get an idea on how to expand the models already present for future works. The same FFNN model, explained in section 3.2.3, and a simple 20-top correlated features selection will be used on a single domain to see if the different essence of faults cause models not to cope with the detection anymore.

Transient Fault In short transient faults happens when a bit is flipped due to cosmic rays hitting a flip-flop.
The best model on validation set is able to reach 1.00 precision and 0.37 recall after 50 epochs on the transient fault test set.

Stuck-at-1 fault Stuck-at-1 fault are the opposite as stuck-at-0, the fault is permanent and the bit is *stucked* at 1.
The best model on validation set is used on test set and it reaches 0.90 precision and 1.00 recall after 50 epochs on the transient fault test set.

All faults together After these results the datasets regarding the basic-math binary with the all three different faults (stuck-at-1, stuck-at-0 and transient) have been merged, the performance on test set (1.00 precision and 0.45 recall) by a model trained for 50 epochs on train set suggests the effects on the architectural state caused by a stuck-at-1 or stuck-at-0 or even transient faults seems to be the same for the same binary.
More data can be collected to work on more general models, able to distinguish from fault type to fault type and maximizing the recall and these results suggests it is feasible.

3.5 Conclusion

In conclusion, for the amount of data that were collected the best performing model is the FFNN with transfer learning for each domain.
Not only it reaches the best performances, Figure 3.16, on each domain but it is also the simplest and it works with 20 features. The small features selection is crucial considering the next chapter will be about an implementation of the whole monitor architecture.
In addition it is important to notice the precision before transfer learning of the SSAE architecture could mean something in the ability of the model of generalizing for many domains. However, for the data that were collected for the goal of this work a result better than that has not been obtained but it needs further investigation in the future.

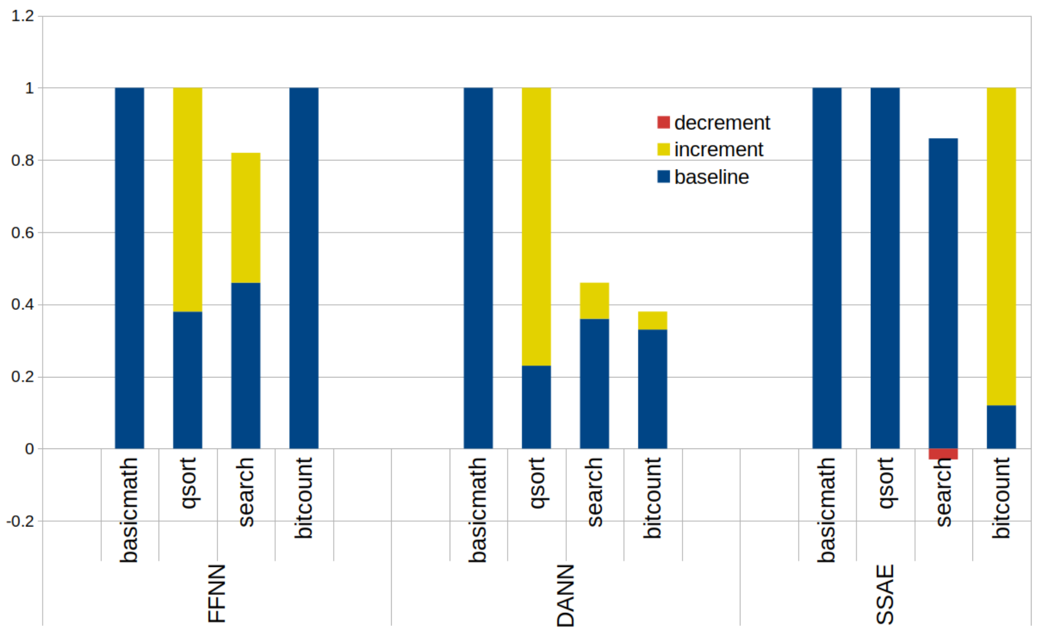


Figure 3.16. Test set precision scores on different binaries from different models before and after the training on the specific domain. Basicmath is used to build the baseline

Chapter 4

OS implementation

In this chapter, the focus will be to present a way to implement inside a Linux distribution.
Before entering the details of the monitor application, the general functioning of the Linux will be displayed along with the development of a kernel module. After that, the PMCTrack, a kernel module for hardware metrics inspection will be analyzed in-depth to show potentiality and possibility of future expansion. Finally, some proposed architecture will be shown.
It is important to highlight this chapter is part of an attempt to show what models can be used. For the intrinsically limited time of this work it is shown the path suggested to follow to further implement a working monitoring system inside an OS, but the resulting product is not complete and it needs hardware support to add essential metrics to be inspected and developers work on the kernel module.

4.1 Introduction

Each program to run needs at least a processor and memory system to run, so a lot of requests toward hardware units are generated, there must be a mechanism on top of them in charge of handling these requests most efficiently.

4.1.1 Linux Kernel

The Linux Kernel [46] is, indeed, the mechanism to do so and much more. The Linux kernel is monolithic with modular design: monolithic means the

entire operating system runs at kernel level and the modular design implies the possibility to add functionalities run time, more on this later **??**.
The Linux kernel main features are:

Memory management each variable in each program needs a place to be stored, this place needs to be fast to access. Fast memory is of course not free and not infinite, so there must be a way to manage this resource. The kernel, then, is in charge of allocating and freeing memory for programs. Specifically, the Linux kernel implements a paged virtual memory, which is an abstraction where the programs see a very large portion of memory and access it using virtual addresses, these addresses will be then converted to access the physical memory. This technique improves the isolation of processes because it avoids having to share memory between processes.
Also, with paging, each program can 'use' more memory than the physically available because some pages can be store in secondary memory, and will be the kernel in charge of keep available in the principal memory the pages used.

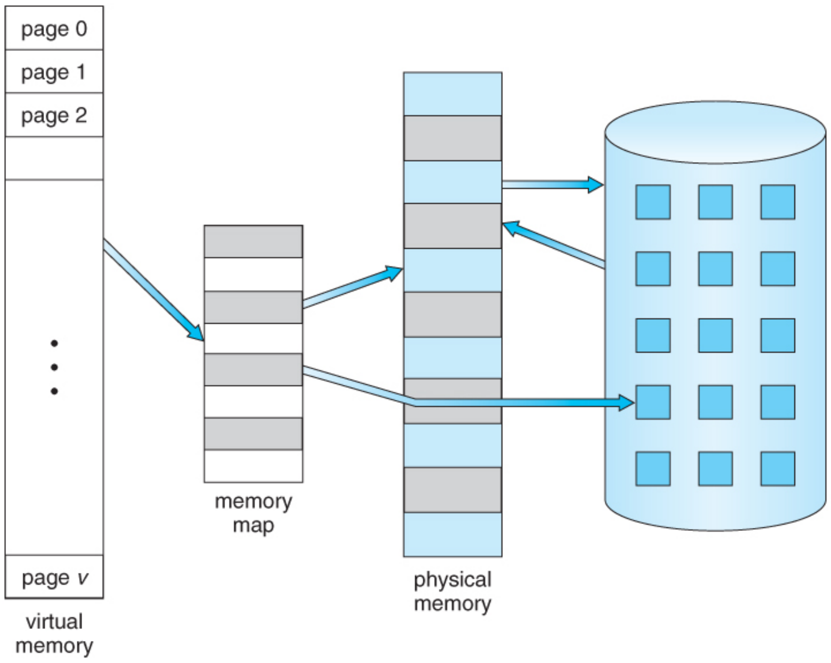


Figure 4.1. Virtual addresses are converted and the corresponding page is searched in principal memory. If there is it is swapped from the secondary memory to the principal memory. Figure from [14]

Process management Along with programs there are processes: a process is the instance of a computer program that is being executed by one or many threads. The scheduler is in charge of scheduling threads, which are a small set of instructions that belonged to a process. Of course, Linux kernel supports multi-threads process, multi-task preemptive scheduling, and concurrent computing (true parallel execution in case of multiple CPUs).
The Linux kernel default scheduler is the Completely Fair Scheduler [26], which implements the idea of assigning each task a period that is proportional to its weight (execution time) divided by the total weight of all the runnable tasks.

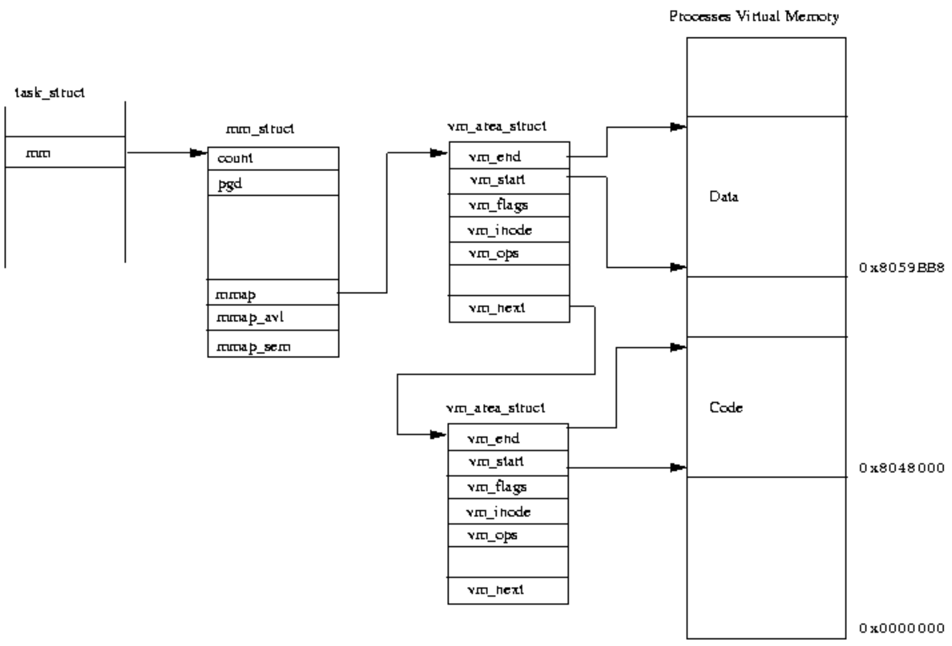


Figure 4.2. The task_struct contains information about the process, the code, the data, and their relative addresses in the virtual memory. Figure from [18]

Device drivers Devices are pieces of hardware attached to the processor, and they communicate with the CPU through a bus. Access to them has to be regulated to avoid conflicts and maximize efficiency. A device driver is an interface to the hardware, that can be used either from kernel or user space to access the functions offered by the hardware or to handle the interrupt

requests coming from it.
They are strongly hardware-dependent and operating system specific.

System calls Usually in computing systems there are 2 running levels: the first is the user level, where applications live and they cannot access the hardware directly. The second is kernel-level or privileged level and it has unrestricted access to hardware, such as scheduling on CPU, disk, or principal memory. This multi-layered architecture is needed to protect the system from user-level applications harming other applications or even the entire system, so the kernel exposes system calls to permits user applications to transfer the control to the kernel to execute the privileged tasks, like allocating a variable.

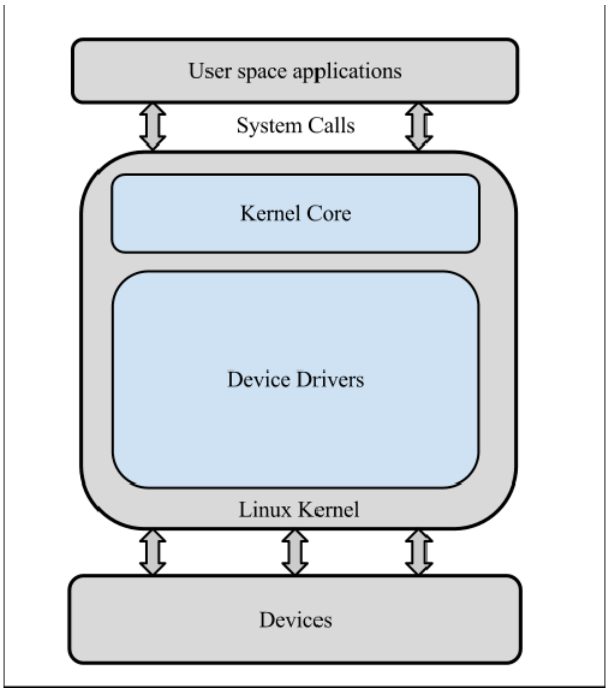


Figure 4.3. Device driver works at kernel level and they are a bridge towards hardware. Systems call serves the purpose of connecting user-level application to the kernel to perform privileged operations. Eg. to access the standard output

4.1.2 Linux Kernel Development

All of these high-level concepts must be translated into C code. Since Linux is an open-source project it is possible to access the code, make some changes, build the kernel with a C compiler and finally run the modified version of the kernel.
It seems easy, however, the Linux kernel has approximately 28 million lines of code [1], so developers needed a way to develop using the kernel API yet avoiding rebuilding the kernel.
This brought, in 1995, the concept of Loadable Kernel Modules, which are pieces of code added while the system is running. They work at the kernel-level and they are used for device drivers, system calls, and filesystem drivers. They are very convenient for different reasons [38]:

- They are as fast as the functions would have been implemented inside the kernel and they can make use of in-kernel API, which are kernel functions exposed especially for kernel modules. Still, the modular approach is in contrast with the monolithic Linux kernel because it breaks the idea of having the kernel on contiguous memory space and degrades a bit the performance because there are more TLB misses.
- The change is not bounded to the kernel, so it is easier to debug. This is because the module is loaded after the system is already booted, so if there is a problem in the module is easier to track.
- They save memory and time because they are loaded only if they are needed, and the building time is negligible with respect to building the entire kernel.
- Even it is suggested to have a good understanding of the kernel, writing a kernel module does not require to have full control of memory management, scheduling, or low-level interaction with hardware. The approach is closer to implementing user-level applications than changing the kernel itself.
- It is easy for user applications to interact with kernel modules thanks to the `/proc` file system. It is possible to declare an entry in the file system that is treated just like a file, so, knowing the file name (Ex. `/proc/monitor`) `open()`, `read()`, `write()` operations can be performed. It is possible to declare functions to handle these interactions in different ways, for example, adding a PID to a list in a kernel module when the process performs a write on the `/proc/monitor` file.

- The only 'disadvantage' is that, since they are implemented at kernel level, they can easily crash the entire system because there are not the protections reserved for user-level programs.

4.2 PMCTrack

«Hardware performance monitoring counters (PMCs) have proven effective in characterizing appli- cation performance.» [36] In this paper Saez et al. demonstrate how the scheduler can be improved if it is fed with high-level information about applications like cache hit ratio or energy consumption. To do so they develop a complex monitoring system that is gonna be analyzed and explained in this section to better understand how to use this infrastructure for another task: fault detection.

4.2.1 PMC

Performance Monitoring Counters are present practically in all modern computing systems, they are special-purpose registers used to keep track of events (i.e. count the number of cache hit).
These hardware counters are used for performance fine-tuning and this is becoming increasingly important for two reasons. The first is to save money: having a 2% faster machine when a company is dealing with many computing systems can be a huge saving of time. The second is related to the nature physic of modern processing units: it has been demonstrated they work better in certain thermal and voltage conditions [33]. However, inspecting these registers is not trivial because their implementations differ from CPU to CPU.

4.2.2 Architecture

PMCTrack is designed following 3 principles, Figure 4.4: exposing high-level metrics to the kernel (i.e. Hit rate per thread), exposing metrics to user-level applications or collect data to be inspected offline by developers as well (i.e. energy consumption per application), and maintaining an architecture-independent core.

Metrics extraction PMCTrack developers wanted to decouple hardware metrics extraction and hardware metrics utilization. Thus, the *access* to PMCs is hardware-independent from the kernel or user level perspective. Configuration, sampling is configured equally from the programmer side if he is working with ARM or Intel. The specific implementation is hidden behind a common interface, and supporting additional hardware facilities is a matter of implementing the correct behaviors in the interface's callbacks.

Kernel module The whole project is developed into a kernel module, in fact it needs a small minor kernel patch to handle the interaction between the kernel module and the kernel itself, and to change the scheduler. This is needed both to collect per-thread data and to implement kernel PMCTrack API.
The kernel module is composed of:

- *PMCTrack architecture-independent core:* this part is in charge of exposing the interface to access metrics to `libpmctrack`, that will expose again metrics to the PMCTrack command-line tools or PMCTrack GUI. Also, the same interface is used by the in-kernel API, for example, to implement new scheduling policies.
- *PMC access layer:* this layer is to give PMC access to the PMCTrack core. It needs to implement specific configuration, direct extraction and it is where developers need to work if they want to add support for different CPU architectures.
- *Monitoring modules:* this layer has been added later to PMCTrack and it is different from PMCs because it is used to gather information from other than CPU hardware pieces (i.e. dram energy consumption)

4.2.3 Functionalities

This framework has been used for optimizing scheduling policies, optimize programs or even compare pieces of hardware [16, 37, 44], however PMCs as it is demonstrated by the preceding chapters they can be useful to detect faults but there are few considerations to point out.
First of all, the data collected by the gem5 simulations are system-wide, although the simulation environment is built just to run the particular binaries and the operating system (scheduler), so it is a good approximation of per-thread data.

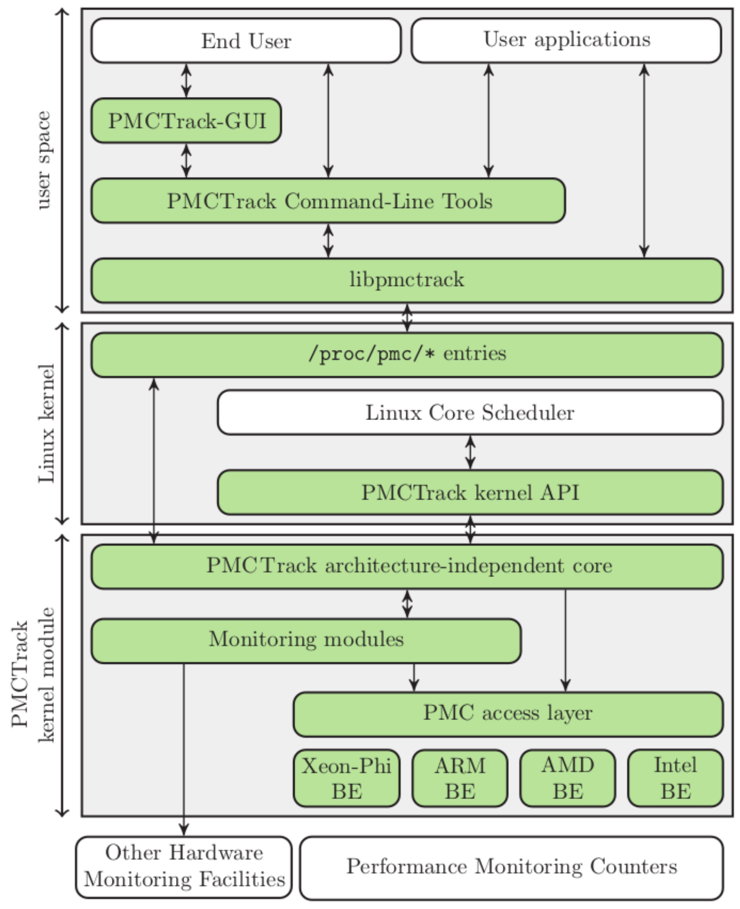


Figure 4.4. PMC Infrastructure: the */proc* entry is used as a broker toward *libpmctrack*, and PMCTrack architecture-independent core is used as a bridge for both user and kernel level access to PMC. Figure from [36].

Secondly, the sampling frequency is different. With simulations, the data were collected after the end of the program, with PMCTrack is decided with a command-line option.
Finally, the number of hardware metrics inspected by PMCTrack, Table 4.1, on a common PC is significantly lower respect to the gem5 simulator.

4.2.4 Limitations

The metrics collection structure is complex and it is easy accessible from PMCTrack interfaces, but it is very difficult to customize and for the moment

PMC	gem5 correspondant	Meaning	ρ
instr_retired_fixed	system.cpu.committedInsts	How many instruction were really exectuted	0.79
unhalted_core_cycles_fixed	system.cpu.num_busy_cycles	Number of cycles puts in HALT by the OS	0.8
instr	sims_insts	The number of ticks simulated	0
cycles	system.cpu.numCycles	Number of CPU cycles simulated	0.81
llc_references	system.cpu.dcache.overall_accesses::total	Number of accesses to cache	0.71
llc_misses	system.cpu.dcache.overall_misses::total	Number of cache misses	0.75
branch_instr_retired	system.cpu.Branches	Number of branch instructions	0.7

Table 4.1. PMCTrack features translated in gem5 metrics with respective correlation with fault labels

it does not have an interface towards other kernel modules.
So in the next section, the architecture will be explained as there is, to show the potentialities of this kind of monitoring system, even if the kernel module communication at the moment is not permitted and it has been bypassed using the PMCTrack user-level libraries.

4.3 Proposed Infrastructures

4.3.1 Completely in-kernel infrastructure

OS/161 [47] is a very simple operating system, with basic memory and process management. So to avoid dealing with the complex kernel, like Linux, OS/161 has been modified to extend the `proc struct` to include some randomly generated metrics and a flag to detect if the process is monitored and if there has been a fault during its execution.
This approach had some interesting features that are useful to cite not to forget if this work will progress:

- The system emulated was multi-processors and the fault detector and OS were scheduled to work on a CPU and the monitored processes on another CPU. This improves the reliability of the system because OS+fault detector and monitored processes are on different CPU so a fault is either compromising processes and the fault detector can see them or it is compromising the OS and this usually generates crashes, so it is not so interesting to detect as it is explained in paragraph 2.2.2.

- The processes flagged as faulty are signaled to the stdout and rescheduled instantly by the fault detector. In this way, it does not need a manual restart each time there is a fault.

- The scheduler is aware of processes monitored and this can be used to modify scheduling procedure.

Unfortunately, OS/161 has a lot of limitations which made the progress impossible. First of all, it does not support `math` library, so it was impossible to support even the simplest machine learning model (e.g. Logistic Regression requires exponential, logistic function, square product). Then it does not have already implemented sockets, so it was also impossible to demand machine learning operations to other OS or machine. Finally, the metrics collection system is really difficult to implement because it is run on a QEMU simulator and it does not have the permission to access directly hardware.
For all these reasons, it has been decided to pass to Linux kernel to have a full-featured kernel and PMCTrack to demand the metrics collection to an already implemented module.

4.4 Kernel Module Infrastructure

The preceding framework would have full control on everything, from PMCs collection to scheduling, however, it needs a massive work and it is usually better to use an already existing platform and build on them despite having to invent the wheel each time.
So in this section, there will be explained how PMCTrack is used to extract metrics and how these metrics are redirected to perform the fault detection tasks.

4.4.1 Kernel Module Facilities

In this little section, different tools exposed by the kernel API will be explained because they are used later on.

RCU List Read, Copy, Update [34] is a synchronization mechanism optimized for read-only situation. The idea is to split the update phase into removal and reclamation.
The *removal* phase references the data and it can be accessed concurrently by readers. It does not generate problems because modern CPU guarantees the reader either read the old or the new version of the referenced data (temporary or intermediate data are read).
The *reclamation* phase is in charge of free data in a structure, and it is executed once all active removal phases are terminated.

The main advantage is that the readers do not require strong synchronization methods, because the writer waits for old readers to finish before updating the data and new readers cannot access old values. So, this algorithm is excellent in the case of read-mostly data, and in many situations is real-time responsive because it does not even need a synchronization method.

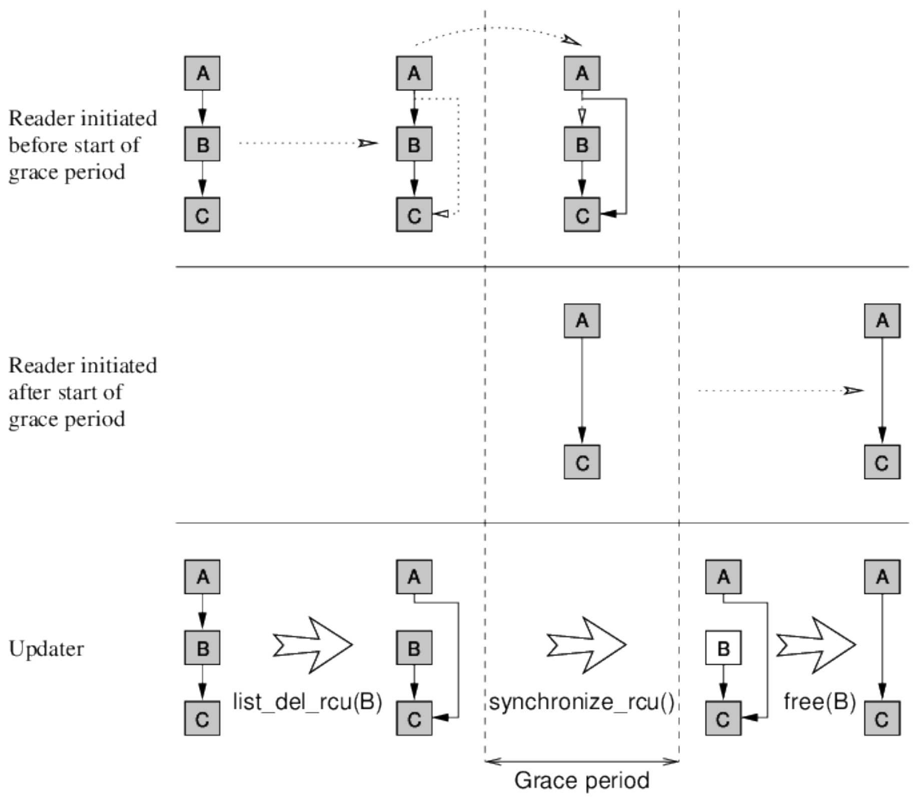

Reader initiated before start of grace period
Reader initiated after start of grace period
Updater
A
B
C
list_del_rcu(B)
synchronize_rcu()
free(B)
Grace period

Figure 4.5. RCU update: the two readers read two different versions of the lists, the first without waiting and the second waiting only the grace time. The difference between RCU and lock-based approach is that the first reader would have been locked out for the whole updating time. Figure from [12].

Socket interface Inside the Linux kernel is even possible to access to a socket interface [27]. This is another way of contacting user-level applications, however, it must be handled carefully because it lacks the support conceived to user-level applications, so configuration, sending, and receiving messages is much more complex, even if it is possible to have TCP support.

Work queue Work queue [] are special structure offered by the kernel API to declare special functions that will be later executed by special kernel threads, these functions will be run in process context so they can block for I/O operations or `wait` operations.
The worker thread most of the time will sleep and it will wake up when work will be added to the work queue. The work struct contains the function to be executed and it can even enclose some data.

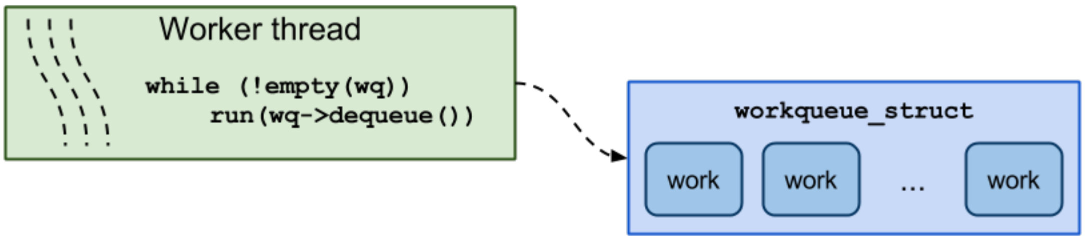


Figure 4.6. Worker thread and work queue: each work struct contains the function to be executed and other information (i.e. values, pointers). Figure from [28].

4.4.2 Monitoring system

The monitoring system, Figure 4.8 is composed making use of all the components and facilities explained before:

- *PID RCU list:* this list contains the PID of processes signaled to be monitored and their respective last features extracted.
- *Update feature work queue:* this work queue is in charge of updating the features related to a PID entering the list. The evaluation work is invoked after a PID has been newly inserted or after it has been evaluated and it resulted non-faulty. This is done to avoid updating the metrics many times before the evaluation process has taken place.
  This part has interaction with the PMCTrack cite in section 4.2.4.
- *Evaluate process work queue:* this queue has at least a work as long as there is a PID in the list. Its job is to extract the features from the PID lists, and demand the job to another queue to send the features thought the socket, wait for the response, and update the PID list accordingly (reinsert the PID if the process is not faulty).
  The evaluation is made user level by a Python program using pre-trained

models, the fact it is contacted using the socket suggests that in the future, for example in a data center, the machine learning evaluations can be made on a predisposed machine reachable from any nodes.

- *Command-line utility:* the command `monitor` is used to start the monitor process PMCTrack side and to inform the monitor module a PID must be added to the list. This is done making use of the `/proc` entry, so executing a write operation on the `/proc/monitor` file with the PID.

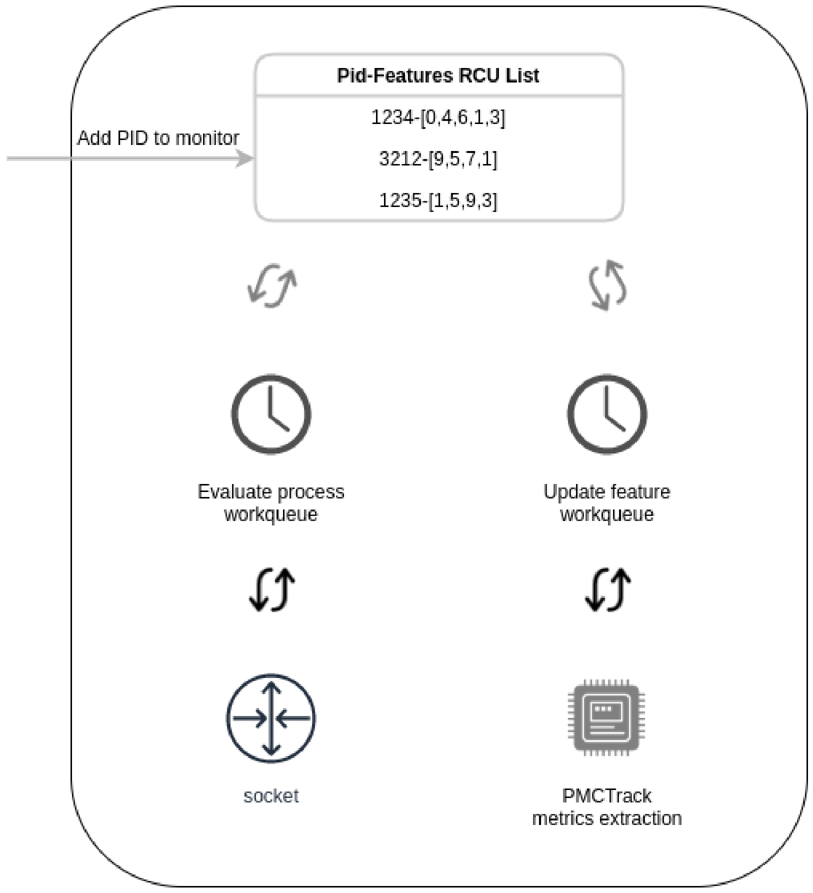


Figure 4.7. Monitoring module

Lifespan of a monitored process:

⇒ **1.** The process is started using the `monitor` process (and this is the only thing the final user has to do).

⇒ **2.** The command signals both the monitoring module and PMCTrack to start to monitor the process.

⇒ **3.** After a configurable (inside the kernel module) lap of time the features about the process are extracted and put into the RCU list.

⇒ **4.** After another configurable lap of time the features extracted are extracted and forwarded to the ML user-level process which responses with 0 (non-faulty) or 1(faulty).

⇒ **5.** if the process is faulty the kernel module is informed and it can do whatever it wants. If not the control restarts from point **3**.

Figure 4.8 is the monitoring system inserted in a common computer to show how it does not interact with other running processes and the kernel, beside a minor patch, is the common Linux kernel.
Performance wise the whole infrastructure is design to leverage the PMCTrack optimized metrics extraction and work on them without affecting further (PMCTrack adds an overhead on tracked process because it needs to collect the metrics) the monitored process and have little impact on the operating system.

4.5 User-level Infrastructure

Another way of implementing the monitor process would have been to implement all at user-level using the `libpmctrack`. However, this very simple approach has the disadvantage of not being predisposed to additional expansion or optimization because it relies on a continuous switching between user and kernel space.

4.6 Experiments

Just to demonstrate the concept, some minor experiments has been tried. In fact, it is possible to simulate a fault in the program counter very easily.

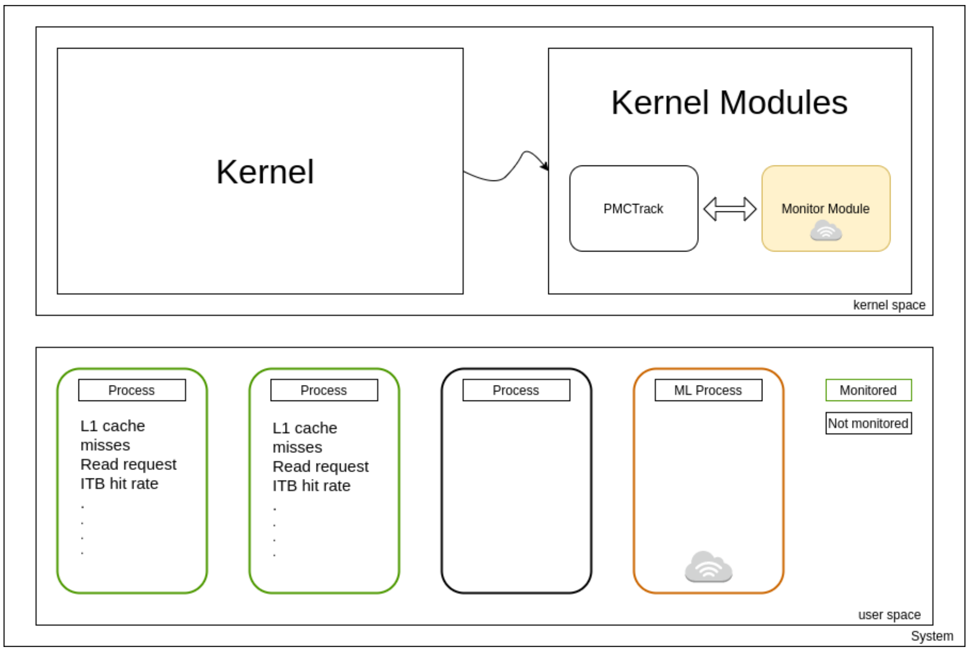


Figure 4.8. Monitoring system: this is the system as it will be on a common computing machine. Remember that the ML process can be even on another machine

```c
int ptr1, ptr2, i = 1000;
ptr1 = &i;
ptr2 = (int *) ((uintptr_t) ptr1 | (uintptr_t) 0xFFF000);
// the fault rate is 50%
if((rand()%2)==0)
  i = *ptr2;
else
    i = *ptr1;
```

Listing 4.1. C program that simulates a fault in its PC during the execution

The resulting PMCTrack metrics are notably different, Figure 4.9, and this is interesting in sight of what machine learning models could do in a different more complex scenario with more metrics.

4.7 Conclusion

This chapter is an analysis on frameworks created and possibly doable in terms of fault detection monitoring system.
The best option for simplicity and performance seems to be the monitoring

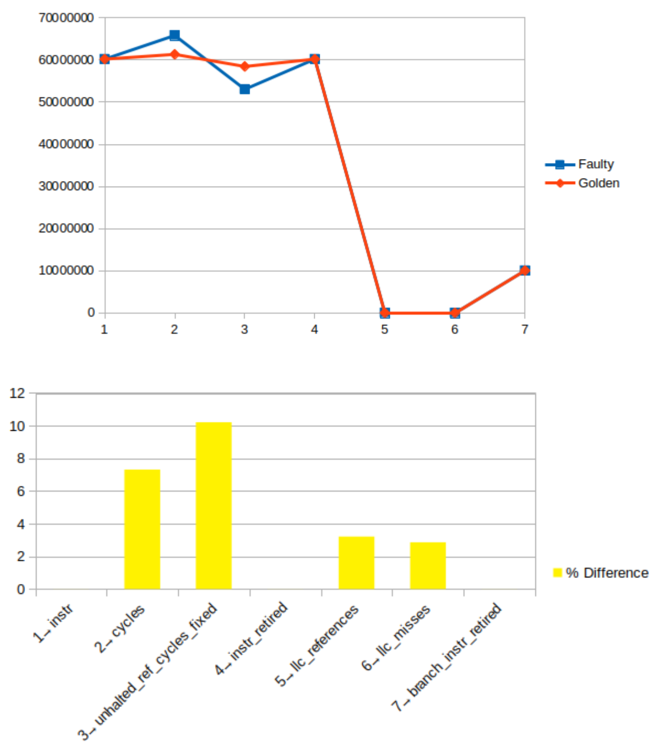


Figure 4.9. Difference between metrics evaluation on faulty and non-faulty run, the second figure gives the % difference to give a scale-relative view

kernel module in combination with the PMCTrack. It is relatively easy to set up, simple for the final user, expandable in the future and it relies on a well-known metrics collection system. However, it needs some work on the interface between PMCTrack and kernel modules, and to be fully functional it needs support for other metrics to use the models to their full extent.

Chapter 5

Conclusion and Future Works

All this project concludes that it is possible to detect faults using hardware metrics, and it is even possible to do so without designing complex model architectures. However, a lot of work needs to be done before the whole system can work reliably, and in this final chapter, some hints for future works will be presented. This part has the goal to be a call for actions to pursue the final goal of this project: to build a system able to detect any type of fault and take consequent responses.
As the thesis is divided into three chapters there will be presented improvements for any of this chapter.

Simulation Environment Nowadays data are the core of each analysis, without them it is impossible to make fundamental steps in any direction and this project is not different. The analysis in 2 involved mainly *stuck-at-0* faults, even if there was a minor experiment to demonstrate these models can be applied to other types of faults, it is safe to say it is needed a broader analysis. In fact, given the nature of transient faults, it is suggested to investigate the problem with many more runs than the ones collected for this work.
In addition to that, it would be important to study the correspondence between PMCTrack and gem5 metrics. This is very useful to understand if the approximation made in Section 4.2.3 holds for different processor architectures or fault types. This requires to work with gem5 internal structure and adapt the actual Linux kernel in use to work with PMCTrack.
Last because it requires modifying the checkpointing system of gem5, it would

be interesting to organize the output as a stream of data from the simulated hardware. This permits the usage of models that will be explained in the next paragraph.

Data analysis and Models After having collected other data, it would be interesting to study different and more complex models to cope with different binaries. For instance, as suggested in the paragraph before, if the output is organized as a stream it is possible to put in place models able to work with time, Recurrent Neural Network [41], or Spiking Neural Network [30]. These models could obtain much better results in detecting faults without specializing on a single binary, so it is worth the analysis.
Also, it is important to evaluate other features selection. Maybe reducing again the number of features, or maybe expanding the number of features with data augmentation techniques to find the minimal set of attributes to simplify the metrics extraction mechanism.

OS Implementation This paragraph is the most important because it contains the improvements needed to implement the architecture described in Chapter 4.
It is needed to expand the number of metrics supported by the kernel module and it is a work that involves both hardware and kernel development.
After that it might be insightful to see the whole architecture in action on the simple C program implemented in Section 4.6.
Also, there is a fascinating field of study that tries to bring machine learning to work inside the kernel [2], so in the future, it could be possible to have a self-contained kernel module able to extract and evaluate metrics without relying on space-level applications.
At last, a proper analysis from the performance point of view is mandatory to ensure the whole framework (PMCTrack or whatever module) is not too heavy on the operating system.

In conclusion, what matters is that this field is engaging and it is stimulating the opportunity of enhancing the current state-of-art operating systems by improving the reliability, which is becoming one of the keystones of automotive, space, or cryptography applications.

Bibliography

[1] Swapnil Bhartiya. *Linux in 2020: 27.8 million lines of code in the kernel, 1.3 million in systemd.* 2020. URL: https://www.linux.com/news/linux-in-2020-27-8-million-lines-of-code-in-the-kernel-1-3-million-in-systemd/.

[2] Ibrahim Umit Akgun, Ali Selman Aydin, and Erez Zadok. «KMLIB: TOWARDS MACHINE LEARNING FOR OPERATING SYSTEMS». In: ().

[3] Robert Baumann. «Soft errors in advanced computer systems». In: *IEEE Design & Test of Computers* 22.3 (2005), pp. 258–266.

[4] Nematollah Bidokhti. «SEU concept to reality (allocation, prediction, mitigation)». In: *2010 Proceedings-Annual Reliability and Maintainability Symposium (RAMS)*. IEEE. 2010, pp. 1–5.

[5] Nathan Binkert et al. «The Gem5 Simulator». In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: https://doi.org/10.1145/2024716.2024718.

[6] Andrea Borghesi et al. «Online anomaly detection in hpc systems». In: *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE. 2019, pp. 229–233.

[7] Leo Breiman. «Random forests». In: *Machine learning* 45.1 (2001), pp. 5–32.

[8] *Bringing Parallelism to the Web with River Trail.* URL: http://intellabs.github.io/RiverTrail/tutorial/.

[9] Athanasios Chatzidimitriou, George Papadimitriou, and Dimitris Gizopoulos. «HealthLog monitor: A flexible system-monitoring Linux service». In: *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*. IEEE. 2018, pp. 183–188.

[10] Yingyi Chen et al. «Application of fault tree analysis and fuzzy neural networks to fault diagnosis in the internet of things (IoT) for aquaculture». In: *Sensors* 17.1 (2017), p. 153.

[11] Cristian Constantinescu. «Trends and challenges in VLSI circuit reliability». In: *IEEE micro* 23.4 (2003), pp. 14–19.

[12] Mathieu Desnoyers et al. «Supplementary Material for User-Level Implementations of Read-Copy Update». In: *MONTH* (Jan. 2010).

[13] Kunihiko Fukushima. «Neocognitron: A hierarchical neural network capable of visual pattern recognition». In: *Neural networks* 1.2 (1988), pp. 119–130.

[14] Peter B Galvin, Greg Gagne, Abraham Silberschatz, et al. *Operating system concepts*. John Wiley & Sons, 2003. Chap. 9.

[15] Yaroslav Ganin et al. «Domain-adversarial training of neural networks». In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 2096–2030.

[16] Adrian Garcia-Garcia, Juan Carlos Saez, and Manuel Prieto-Matias. «Contention-aware fair scheduling for asymmetric single-isa multicore systems». In: *IEEE Transactions on Computers* 67.12 (2018), pp. 1703–1719.

[17] M. R. Guthaus et al. «MiBench: A Free, Commercially Representative Embedded Benchmark Suite». In: *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*. WWC '01. USA: IEEE Computer Society, 2001, 3–14. ISBN: 0780373154.

[18] Greg Hankins. *Linux Documentation Project*. Linux Documentation Project (LDP), 2001. Chap. 4.

[19] Sergey Ioffe and Christian Szegedy. «Batch normalization: Accelerating deep network training by reducing internal covariate shift». In: *arXiv preprint arXiv:1502.03167* (2015).

[20] Jeremy Jordan. *Introduction to autoencoders*. 2018. URL: https://www.jeremyjordan.me/autoencoders/.

[21] Jeremy Jordan. *Setting the learning rate of your neural network*. 2020. URL: https://www.jeremyjordan.me/nn-learning-rate/.

[22] Diederik P Kingma and Jimmy Ba. «Adam: A method for stochastic optimization». In: *arXiv preprint arXiv:1412.6980* (2014).

[23] Kevin Lavery. «Discriminating Between Soft Errors and Hard Errors in RAM». In: *SPNA109*. 2008.

[24] Dean Lee et al. «Convolutional neural net and bearing fault analysis». In: *Proceedings of the International Conference on Data Mining (DMIN)*. The Steering Committee of The World Congress in Computer Science, Computer . . . 2016, p. 194.

[25] Ki Bum Lee, Sejune Cheon, and Chang Ouk Kim. «A convolutional neural network for fault classification and diagnosis in semiconductor manufacturing processes». In: *IEEE Transactions on Semiconductor Manufacturing* 30.2 (2017), pp. 135–142.

[26] 05 Feb 2019Marty Linux Kernel Documentation. *CFS: Completely fair process scheduling in Linux*. URL: `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`.

[27] *Linux Kernel Networking*. URL: `https://linux-kernel-labs.github.io/refs/heads/master/labs/networking.html`.

[28] Vita Loginova. *Multitasking in the Linux Kernel. Workqueues*. 2015. URL: `https://kukuruku.co/post/multitasking-in-the-linux-kernel-workqueues/`.

[29] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. «Rectifier nonlinearities improve neural network acoustic models». In: *Proc. icml*. Vol. 30. 1. 2013, p. 3.

[30] Piotr S Maciąg et al. «Unsupervised Anomaly Detection in Stream Data with Online Evolving Spiking Neural Networks». In: *arXiv preprint arXiv:1912.08785* (2019).

[31] Ibomoiye Domor Mienye, Yanxia Sun, and Zenghui Wang. «Improved sparse autoencoder based artificial neural network approach for prediction of heart disease». In: *Informatics in Medicine Unlocked* (2020), p. 100307.

[32] Shubhendu S Mukherjee, Michael Kontz, and Steven K Reinhardt. «Detailed design and evaluation of redundant multi-threading alternatives». In: *Proceedings 29th annual international symposium on computer architecture*. IEEE. 2002, pp. 99–110.

[33] George Papadimitriou et al. «Harnessing voltage margins for energy efficiency in multicore CPUs». In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 2017, pp. 503–516.

[34] *RCU List.* URL: https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt.

[35] Yara Rizk et al. «Deep belief networks and cortical algorithms: A comparative study for supervised classification». In: *Applied Computing and Informatics* 15.2 (2019), pp. 81–93.

[36] J. C. Saez et al. «PMCTrack: Delivering Performance Monitoring Counter Support to the OS Scheduler». In: *The Computer Journal* 60.1 (Jan. 2017), pp. 60–85. ISSN: 0010-4620. DOI: 10.1093/comjnl/bxw065. eprint: https://academic.oup.com/comjnl/article-pdf/60/1/60/10329287/bxw065.pdf. URL: https://doi.org/10.1093/comjnl/bxw065.

[37] Juan Carlos Saez et al. «On the interplay between throughput, fairness and energy efficiency on asymmetric multicore processors». In: *The Computer Journal* 61.1 (2018), pp. 74–94.

[38] Peter Jay Salzman, Michael Burian, and Ori Pomerantz. *The linux kernel module programming guide.* 2007. Chap. 2.

[39] Jürgen Schmidhuber. «Deep learning in neural networks: An overview». In: *Neural networks* 61 (2015), pp. 85–117.

[40] Rajalingappaa Shanmugamani. *Deep Learning for Computer Vision: Expert techniques to train advanced neural networks using TensorFlow and Keras.* Packt Publishing Ltd, 2018.

[41] Alex Sherstinsky. «Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network». In: *Physica D: Nonlinear Phenomena* 404 (2020), p. 132306.

[42] Leslie N Smith. «Cyclical learning rates for training neural networks». In: *2017 IEEE Winter Conference on Applications of Computer Vision (WACV).* IEEE. 2017, pp. 464–472.

[43] Nitish Srivastava et al. «Dropout: a simple way to prevent neural networks from overfitting». In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[44] Ashraf Suyyagh and Zeljko Zilic. «Real-time benchmark set synthesis based on pWCET estimation and bounded hyper-periods». In: *2017 International Conference on Circuits, System and Simulation (ICCSS).* IEEE. 2017, pp. 129–133.

[45] Prasanna Tamilselvan and Pingfeng Wang. «Failure diagnosis using deep belief learning based health state classification». In: *Reliability Engineering & System Safety* 115 (2013), pp. 124–135.

[46] *The Linux Kernel Archives*. URL: https://www.kernel.org/.

[47] *The OS/161 Instructional Operating System*. URL: http://os161.eecs.harvard.edu/.

[48] Markus Thill, Wolfgang Konen, and Thomas Bäck. «Online anomaly detection on the webscope S5 dataset: A comparative study». In: *2017 Evolving and Adaptive Intelligent Systems (EAIS)*. IEEE. 2017, pp. 1–8.

[49] Tiedo Tinga. «Application of physical failure models to enable usage and load based maintenance». In: *Reliability Engineering & System Safety* 95.10 (2010), pp. 1061–1075.

[50] *Understanding, Building and Using Neural Network Machine Leaning Models using Oracle 18c*. URL: https://developer.oracle.com/databases/neural-network-machine-learning.html.

[51] Gulay Yalcin et al. «FIMSIM: A fault injection infrastructure for microarchitectural simulators». In: *2011 IEEE 29th International Conference on Computer Design (ICCD)*. IEEE. 2011, pp. 431–432.

[52] Jason Yosinski et al. «Understanding neural networks through deep visualization». In: *arXiv preprint arXiv:1506.06579* (2015).

[53] Cheng Zhang et al. «A method of fault diagnosis for rotary equipment based on deep learning». In: *2018 Prognostics and System Health Management Conference (PHM-Chongqing)*. IEEE. 2018, pp. 958–962.