POLITECNICO
MILANO 1863

System modeling with gem5

Giuseppe Calcagno

May 2021

Abstract

Ensuring the safety of an electronic device during its execution is a very important issue, especially in cases where the design and manufacture are carried out by different companies. This project aims to simulate and monitor some cryptographic algorithms on different hardware architectures with the use of gem5 simulator[1] .

Contents

1 Introduction

1.1 Environment Setup

The gem5 simulator requires a Linux environment, so I created a virtual machine using Oracle VM VirtualBox[3]. Gem5 is supported and tested periodically on Ubuntu 18.04 and Ubuntu 20.04, so i decided to install **Ubuntu 18.04** [4] to have greater certainties on the stability of the packages. After the installation of the OS i followed the Gem5 Technical Guide [1] to install the software:

- I have installed all the prerequisites reported using the command:

```
sudo apt install build-essential git m4 scons zlib1g zlib1g-dev
libprotobuf-dev protobuf-compiler libprotoc-dev libgoogle-perftools-dev
python-dev python
```

- I downloaded the source code from the official repository:

```
git clone https://gem5.googlesource.com/public/gem5
```

- I tried to compile the program:

```
python3 scons build/X86/gem5.opt
```

During compilation I found an error: the version of python used to compile was not the correct one. To solve the problem I have set python 3 as the default version of the machine and by doing so I completed the compilation. However at the end of the process I encountered the following warnings:

```
*** Summary of Warnings ***
Warning: Header file <png.h> not found.
         This host has no libpng library.
         Disabling support for PNG framebuffers.
Warning: Couldn't find any HDF5 C++ libraries. Disabling HDF5 support.
```

These warnings were solved with the help of the StackOverFlow community I contacted [5], which pointed out the missing packages. I installed them with the following commands:

```
sudo apt-get install libhdf5-dev libpng-dev
```

At this point, all the warnings were resolved by deleting the old buildings file and compiling gem5 again. Let's start to use Gem5!

1.2 First Step on Gem5

To start I decided to create a simple configuration script that simulated a CPU-Memory bus-memory system following the gem5 manual [2]

Note: Some instructions are different from the gem5 manual, because the one reported is deprecated or incorrect

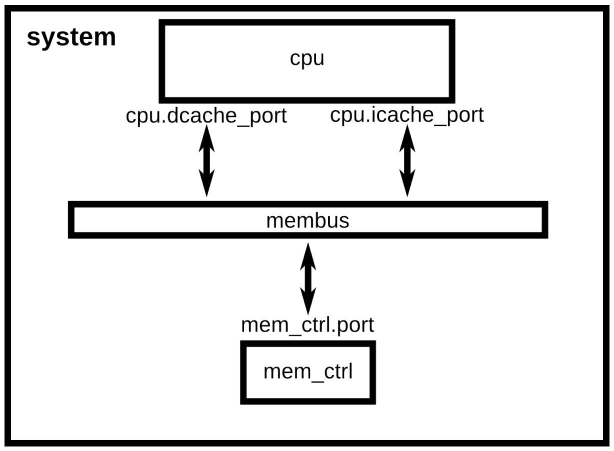


Figure 1: CPU-MemBus-Memory Sistem

I started by creating a new config file in *gem5/configs/tutorial/* called *"simple.py"*. The first thing I'll do in this file is import the m5 library and all SimObjects that I compiled previously.

```
import m5
from m5.objects import *
```

Next, I created the system SimObjects and I configured it.

```
system = System()
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '1GHz'
system.clk_domain.voltage_domain = VoltageDomain()
```

After that, I set up how the memory will be simulated, created the CPU, Memory Bus and Memory system

```
system.mem_ranges = [AddrRange('512MB')]
system.cpu = TimingSimpleCPU()
system.membus = SystemXBar()
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
```

And I connected all the components correctly

```
system.cpu.icache_port = system.membus.cpu_side_ports
system.cpu.dcache_port = system.membus.cpu_side_ports

system.cpu.createInterruptController()
system.cpu.interrupts[0].pio = system.membus.mem_side_ports
system.cpu.interrupts[0].int_requestor = system.membus.cpu_side_ports
system.cpu.interrupts[0].int_responder = system.membus.mem_side_ports
system.mem_ctrl.port = system.membus.mem_side_ports
system.system_port = system.membus.cpu_side_ports
system.mem_ctrl.port = system.membus.mem_side_ports
```

Completed the system, I moved on to instantiate the process to be executed.

```
#this is a "HelloWorld" C programm, present in the gem5 test files
binary = 'tests/test-progs/hello/bin/x86/linux/hello'
system.workload = SEWorkload.init_compatible(binary)
process = Process()
process.cmd = [binary]

system.cpu.workload = process
system.cpu.createThreads()

root = Root(full_system = False, system = system)
m5.instantiate()

print("Beginning simulation!")
exit_event = m5.simulate()
print('Exiting @ tick {} because {}'
      .format(m5.curTick(), exit_event.getCause()))
```

Now I can run my first simulation!

```
build/X86/gem5.opt configs/tutorial/simple.py
```

1.3 Simulation analysis

The output of the simulation just described is:

```
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 21.0.0.0
gem5 compiled May 10 2021 13:51:18
gem5 started May 16 2021 10:47:36
gem5 executing on giuseppe-VirtualBox, pid 2280
command line: build/X86/gem5.opt configs/tutorial/part1/simple.py

Global frequency set at 1000000000000 ticks per second
warn: DRAM device capacity (8192 Mbytes) does not match the
address range assigned (128 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7000
Beginning simulation
info: Entering event queue a 0.  Starting simulation...
Hello world
Exiting a tick 454646000 because exiting with last active thread context
```

In addition to the console output, 3 different files are generated by gem5 in *gem5/m5out*:

- stat: A text representation of all of the gem5 statistics registered for the simulation.
- config: Contains a list of every SimObject created for the simulation and the values for its parameters.
- config.svg: A graphical representation of the instantiated system

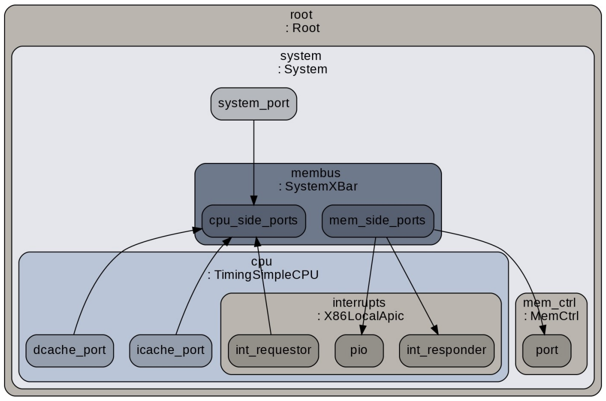


Figure 2: Result of SimpleScript simulation

1.4 Adding Cache to SimpleScript

To improve the system I tried to add 2 levels of cache to the script of the previous paragraph: more precisely I wanted to connect 2 caches (*L1 data cache* and *L1 instruction cache*) to the CPU, connected by another bus to a *L2 cache*, which interfaces with the memory with a bus.

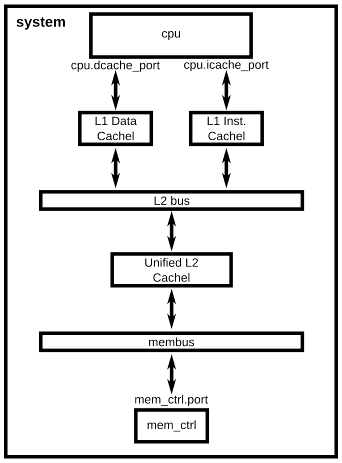
First of all I created a new configuration file where I describe the characteristics of the memories called *caches.py*, in the same directory as *simple.py*:

```
class L1Cache(Cache):
    import m5
    from m5.objects import Cache
    assoc = 2
    tag_latency = 2
    data_latency = 2
    response_latency = 2
    mshrs = 4
    tgts_per_mshr = 20
    def __init__(self, options=None):
        super(L1Cache, self).__init__()
        pass
    def connectBus(self, bus):
        """Connect this cache to a memory-side bus"""
        self.mem_side = bus.cpu_side_ports
    def connectCPU(self, cpu):
        """Connect this cache's port to a CPU-side port
           This must be defined in a subclass"""
        raise NotImplementedError
class L1ICache(L1Cache):
    # Set the default size
    size = '16kB'
    def __init__(self, opts=None):
        super(L1ICache, self).__init__(opts)
        if not opts or not opts.l1i_size:
            return
        self.size = opts.l1i_size
    def connectCPU(self, cpu):
        """Connect this cache's port to a CPU icache port"""
        self.cpu_side = cpu.icache_port
class L1DCache(L1Cache):
    # Set the default size
    size = '64kB'
    def __init__(self, opts=None):
        super(L1DCache, self).__init__(opts)
        if not opts or not opts.l1d_size:
            return
        self.size = opts.l1d_size
    def connectCPU(self, cpu):
        self.cpu_side = cpu.dcache_port
class L2Cache(Cache):
    # Default parameters
    size = '256kB'
    assoc = 8
    tag_latency = 20
    data_latency = 20
    response_latency = 20
    mshrs = 20
    tgts_per_mshr = 12
    def __init__(self, opts=None):
        super(L2Cache, self).__init__()
        if not opts or not opts.l2_size:
            return
        self.size = opts.l2_size
    def connectCPUSideBus(self, bus):
        self.cpu_side = bus.mem_side_ports
    def connectMemSideBus(self, bus):
        self.mem_side = bus.cpu_side_ports
```

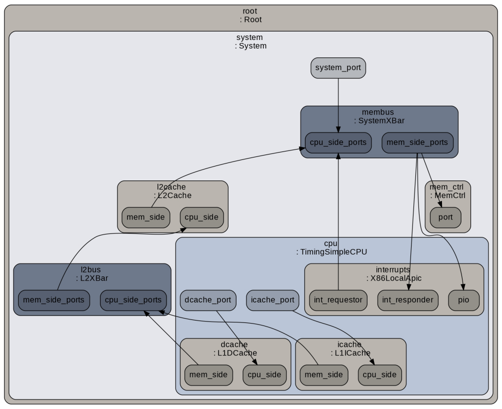
Then I modified the initial script to add caches to the system:

```
class L1Cache(Cache):
import m5
from m5.objects import *
from caches import *
from optparse import OptionParser
parser = OptionParser()
parser.add_option('--l1i_size', help="L1 instruction cache size")
parser.add_option('--l1d_size', help="L1 data cache size")
parser.add_option('--l2_size', help="Unified L2 cache size")
(options, args) = parser.parse_args()

system = System()
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '1GHz'
system.clk_domain.voltage_domain = VoltageDomain()
system.mem_mode = 'timing'
system.mem_ranges = [AddrRange('512MB')]
system.cpu = TimingSimpleCPU()

system.cpu.icache = L1ICache(options)
system.cpu.dcache = L1DCache(options)
system.cpu.icache.connectCPU(system.cpu)
system.cpu.dcache.connectCPU(system.cpu)

system.l2bus = L2XBar()
system.cpu.icache.connectBus(system.l2bus)
system.cpu.dcache.connectBus(system.l2bus)

system.l2cache = L2Cache(options)
system.l2cache.connectCPUSideBus(system.l2bus)

system.membus = SystemXBar()
system.l2cache.connectMemSideBus(system.membus)

system.cpu.createInterruptController()
system.cpu.interrupts[0].pio = system.membus.mem_side_ports
system.cpu.interrupts[0].int_requestor = system.membus.cpu_side_ports
system.cpu.interrupts[0].int_responder = system.membus.mem_side_ports
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.mem_side_ports
system.system_port = system.membus.cpu_side_ports

binary = 'tests/test-progs/hello/bin/x86/linux/hello'
system.workload = SEWorkload.init_compatible(binary)
process = Process()
process.cmd = [binary]
system.cpu.workload = process
system.cpu.createThreads()
root = Root(full_system = False, system = system)
m5.instantiate()
print("Beginning simulation!")
exit_event = m5.simulate()
print('Exiting @ tick {} because {}'
      .format(m5.curTick(), exit_event.getCause()))
```

After running the simulation of the new model, I used the graphical representation of the instantiated system to check that the system matches the theoretical one:



(a) theoretical model



(b) Simulation result

2 Multi Processor System

After simulating a simple system, following the gem5 manual [2], I moved on to simulate a system capable of carrying out multiple processes in parallel, using multiple processors capable of sharing data between them. For memory management I used Ruby: it provides a detailed cache memory and cache coherence models as well as a detailed network model.

2.1 Introduction

First of all, I created the *SConspts* file in *src* folder, which indicates the creation of a new memory management protocol and specify its name.

```
Import('*')
all_protocols.extend([
'MSI',
])
protocol_dirs.append(str(Dir('.').abspath))
```

2.2 State Machine

After declaring the protocol, I need to describe the heart of the whole protocol: the state machine, which is made up of:

- **Parameters** These are the parameters for the SimObject that will be generated. Here 3 necessary parameters are declared: the **sequencer**, which serves to interface the Ruby memory system with the rest of the Gem5 system, the **cacheMemory**, which contains the memory specifications and the **buffers** that interface the state machine with the ruby system.

- **Declaring required structures and functions** This section declares the states, events, and many other required structures for the state machine.

- **In port code blocks** This section contain the code that looks at incoming messages from the (in_port) message buffers and determines the events to trigger. Here we also specify the implementation of the messages that we send through the ports.

- **Actions** These are simple one-effect code blocks that are executed when going through a transition from one state to another. In this phase, the request data to a cache and the sending of data to the forward port functions are implemented and the size of the messages is defined. CPU request handling and message buffer management are also specified here.

- **Transitions** This section specify actions to execute, give a starting and final state to the ASF.

2.3 Other Files

To complete the definition of the consistency protocol, I need to create 3 more files:

- **MSI-msg.sm** where new message types are implemented
- **MSI-dir.sm** where a memory controller is defined
- **MSI.slicc** This file tells the SLICC compiler which state machine files to compile for this protocol.

At this point the protocol is completed, so I moved on to instantiate a system that uses it.

NOTE: The state machine is coded using *SLICC* (Specification Language including Cache Coherence)

2.4 System creation

The creation of the system is not very different from that of the systems seen above, however some additional steps are needed: I defined the number and type of processors to use (4 Timing CPU) and I connected them to the cache system defined in the same way as the previous systems. What changes is the cache controller, which is updated according to the **MSI-dir.sm** file. After setting up the components I initialized the ruby system with the newly created consistency protocol and system. Lastly, I created the network between the various processors (and related memories) with simple point-to-point connections.

Below we find a graphical representation of the system and the network. We note that the system matches the one described, and in the network we find the 4 processor nodes plus the ruby system controller. NOTE: the scripts used are taken from the gem5 source code. The documentation version has some inaccuracies

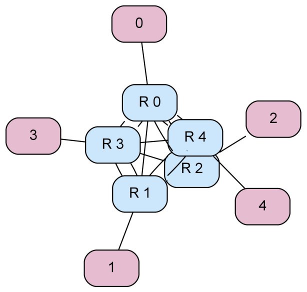


Figure 4: Network Configuration

.

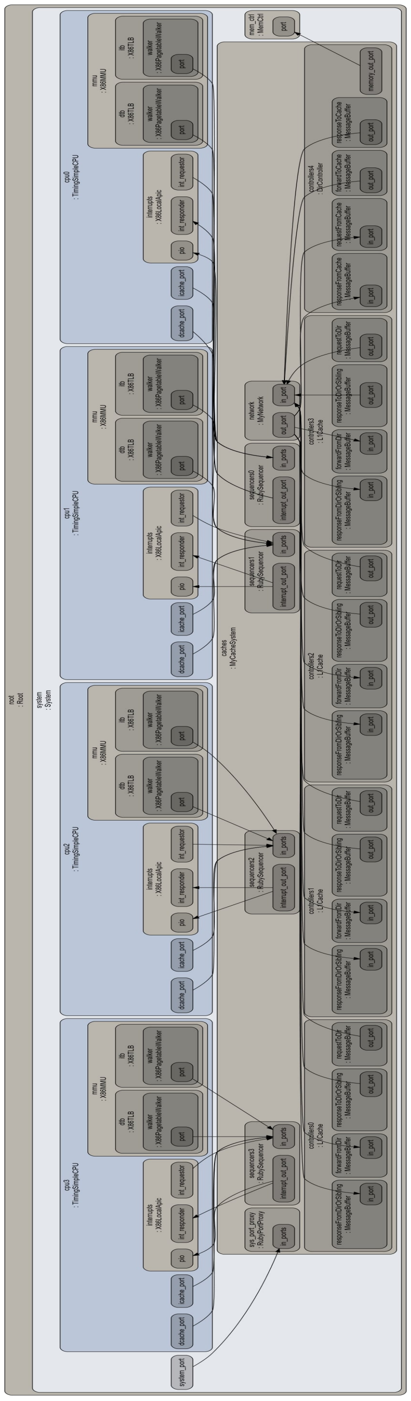


Figure 5: System Configuration

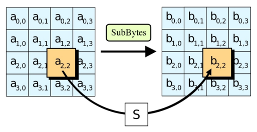
3 AES: advanced encryption standard

In cryptography, **AES** [6] (advanced Encryption Standard) is a symmetric key block cipher algorithm, its standard implementation involves a **substitution** and **permutation** network. it is used as a security standard by the United States government, the National Institute of Standards and Technology and the US FIPS PUB.

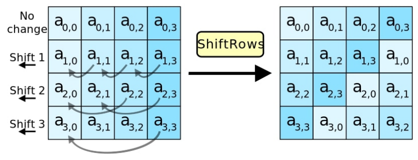
3.1 Description of the algorithm

The data to be encrypted is divided into 128-bit blocks and encrypted with a 128,192 or 256 bit key. The algorithm operates using matrices of 4x4 bytes called states on which the following operations are performed:

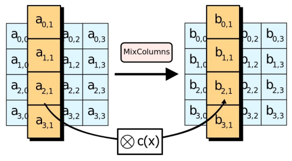
1. **SubBytes:** Non-linear replacement of all bytes. To do this, an 8-bit "S-box" (replacement box) is used. This obscures the relationship between plain text and cipher text.



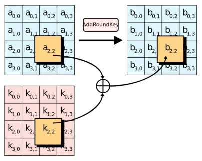
2. **ShiftRow:** This step involves the shift of the elements of the state according to the row to which they belong: the first line remains unchanged, while the others are shifted incrementally.



3. **MixColumns:** Takes the 4 bytes of each column and combines them using an irreversible linear transformation



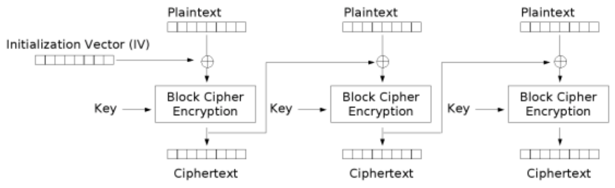
4. **AddRoundKey:** Combine with a xor the session key and the matrix obtained in the previous steps



3.2 Types of Block Encryption

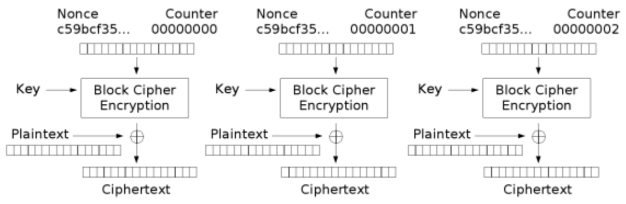
After seeing how each state is encrypted, all that remains to do is to define how to manage the various states. This is done in several ways, those used for this project are:

- **Cipher Block Chaining (CBC)** [7] in this mode what is encrypted is not the state, but the result of the XOR between the state and the previous encoding. An initialization vector is used for the first state



Cipher Block Chaining (CBC) mode encryption

- **Counter (CTR)** [7] In this mode a counter of size equal to the size of the state is used. The counter is encrypted and placed in xor with the plain text to obtain the final encryption. This implementation has a higher efficiency than CBC mode as it is possible to perform pre-processing and parallelize operations without compromising the security of the algorithm.



Counter (CTR) mode encryption

3.3 Test bench Program

To analyze the system performance impact of this type of algorithm I created a program used as a test bench:

I started by selecting a suitable library that would allow me to adequately vary the characteristics of the cryptography, my choice fell on ”tiny-AES-c” [8]: This library allows me to encrypt and decrypt data using an implementation of AES, it also allows me to select the type of block management mode (CBC or CTR) and the length of the keys (128, 192 or 256 bit).

Then I created a program (*AES_MAIN*) that reads a file from disk, the content is encrypted according to both CBC and CTR modes with a desired key length and writes the result to another file, at the end of these steps it decrypts the file and the result is written into the same file. I also modified this program to create a new version (*AES_THREAD*) that would perform the operations in parallel for the 2 modes.

To have a term of comparison, I created another program (*TEMPOFILE*) that reads the initial file, writes a number of letters equal to its size (to simulate the printing of the result) and then copies the content twice to another file (to simulate printing of decryption). In this way I can understand how much time during the execution is dedicated to managing the file and how much time is instead dedicated to encrypt and decrypt the data.

All the source code was written in C and can be found on GitHub [9] along with 2 programs to create a file of desired size and count the number of characters of an existing file.

4 Tests and Simulations

The hardware systems used for the simulations are those described in the previous chapters, but before starting I would like to give some additional information about them

4.1 Characteristics of the Architectures

- **Simple System:** This system is the simplest, the CPU is connected directly to the memory via a bus.

```
CPU: 1 TimingSimpleCPU  Hz= 1Ghz
Memory Controller: DDR3_1600
```

- **Two Level Cache System** In this system we find 2 cache levels that separate the CPU from the memory

```
CPU: 1 TimingSimpleCPU Hz= 1Ghz
L1ICache: dim=16kB  data_latency = 2ns  response_latency = 2ns
L1DCache: dim=64kB  data_latency = 2ns  response_latency = 2ns
L2Cache: dim=256kB  data_latency = 20ns  response_latency = 20ns
Memory Controller: DDR3_1600
```

- **Ruby System:** is the most complex system, it is composed of 4 CPUs connected to the Ruby cache system, composed of 4 caches

```
CPU: 4x TimingSimpleCPU Hz= 1Ghz
Cache: 4x  dim=16kB  data_latency = 2ns  response_latency = 2ns
Memory Controller: DDR3_1600
```

The TimingSimpleCPU uses timing memory accesses. It stalls on cache accesses and waits for the memory system to respond prior to proceeding. All simulations were performed in **timing mode**: *"reflect our best effort for realistic timing and include the modeling of queuing delay and resource contention. Once a timing request is successfully sent at some point in the future the device that sent the request will either get the response or a NACK if the request could not be completed"*

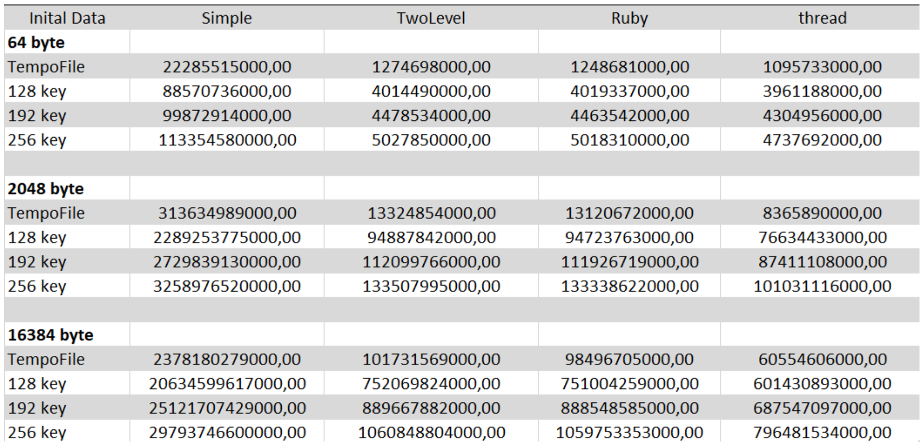
4.2 The tests

On all types of architecture I have decided to run the *AES_MAIN* program using files of size equal to 64, 2048 and 16384 bytes with a key of 128,192 and 256 bits for each type of simulation. Then I ran the *TEMPOFILE* program in the same way (all architectures with file sizes of 64, 2048 and 16384 bytes).

Having more processors available, I used the *AES_THREAD* program to evaluate the performance of the ruby system in the same way as the other test benches.

Finally I decided to run the 2 CBC and CTR modes in separate tests to evaluate the performance differences between. This test is performed on the ruby architecture only

Here are the results I got. All results are expressed in ticks (ticks per second: 10^{12}):



Inital Data	Simple	TwoLevel	Ruby	thread
64 byte				
TempoFile	22285515000,00	1274698000,00	1248681000,00	1095733000,00
128 key	88570736000,00	4014490000,00	4019337000,00	3961188000,00
192 key	99872914000,00	4478534000,00	4463542000,00	4304956000,00
256 key	113354580000,00	5027850000,00	5018310000,00	4737692000,00
2048 byte				
TempoFile	313634989000,00	13324854000,00	13120672000,00	8365890000,00
128 key	2289253775000,00	94887842000,00	94723763000,00	76634433000,00
192 key	2729839130000,00	112099766000,00	111926719000,00	87411108000,00
256 key	3258976520000,00	133507995000,00	133338622000,00	101031116000,00
16384 byte				
TempoFile	2378180279000,00	101731569000,00	98496705000,00	60554606000,00
128 key	20634599617000,00	752069824000,00	751004259000,00	601430893000,00
192 key	25121707429000,00	889667882000,00	888548585000,00	687547097000,00
256 key	29793746600000,00	1060848804000,00	1059753353000,00	796481534000,00

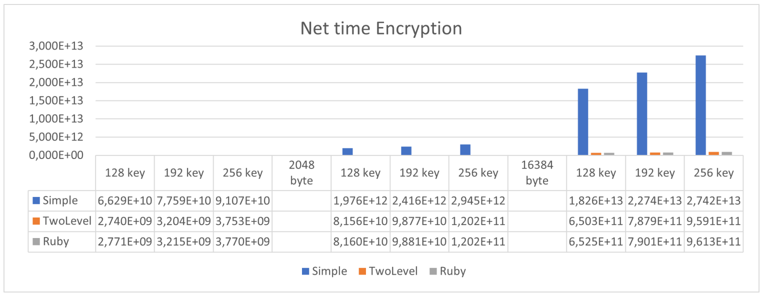
Figure 6: Results of the Simulations

All test bench executables can be found on GitHub in the "Testbench Executables" folder [9]. Note: some tests took about 45 minutes!

5 Analysis of the results

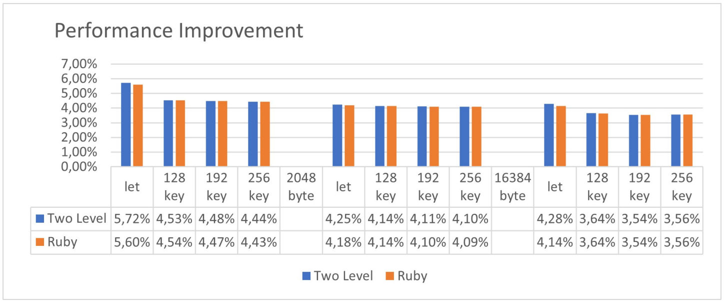
5.1 Architectures performance

I would like to start by focusing on the total times of the simulations on the architectures. As expected, the "*simple*" architecture took much longer than the other architectures despite the fact that the CPU was the same. The lack of cache memories greatly worsened the performance of the architecture which, having to access the main memory, encountered many periods of inactivity.



	128 key	192 key	256 key	2048 byte	128 key	192 key	256 key	16384 byte	128 key	192 key	256 key
■ Simple	6,629E+10	7,759E+10	9,107E+10		1,976E+12	2,416E+12	2,945E+12		1,826E+13	2,274E+13	2,742E+13
■ TwoLevel	2,740E+09	3,204E+09	3,753E+09		8,156E+10	9,877E+10	1,202E+11		6,503E+11	7,879E+11	9,591E+11
■ Ruby	2,771E+09	3,215E+09	3,770E+09		8,160E+10	9,881E+10	1,202E+11		6,525E+11	7,901E+11	9,613E+11

What is not taken for granted, however, is the little difference in performance between the "*Ruby*" and "*Two-Cache-Level*" architecture. For file reading and memory management, the different configuration of the caches favored the multiprocessor architecture despite the smaller size of its cache memory. This is likely due to the higher latency of the concurrent architecture's second cache level. Regarding the computing power, and therefore the encryption times, the time is very similar since in the Ruby architecture the possibility of having parallel processes is not exploited (multi thread data will be provided later). Below is a graph of the percentage improvement of the 2 architectures compared to the *Simple* one.

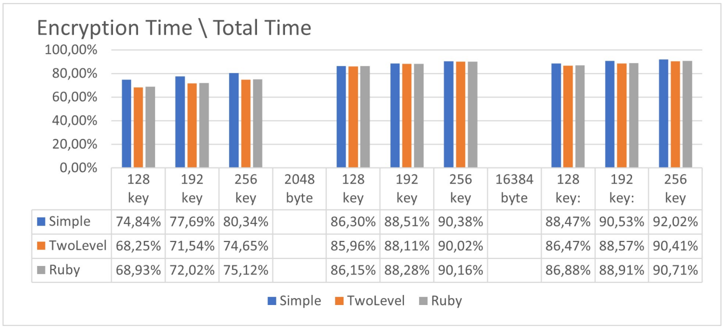


	let	128 key	192 key	256 key	2048 byte	let	128 key	192 key	256 key	16384 byte	let	128 key	192 key	256 key
🟦 Two Level	5,72%	4,53%	4,48%	4,44%		4,25%	4,14%	4,11%	4,10%		4,28%	3,64%	3,54%	3,56%
🟧 Ruby	5,60%	4,54%	4,47%	4,43%		4,18%	4,14%	4,10%	4,09%		4,14%	3,64%	3,54%	3,56%

Figure 7: Percentage improvement

5.2 Relevance of cryptography to system performance

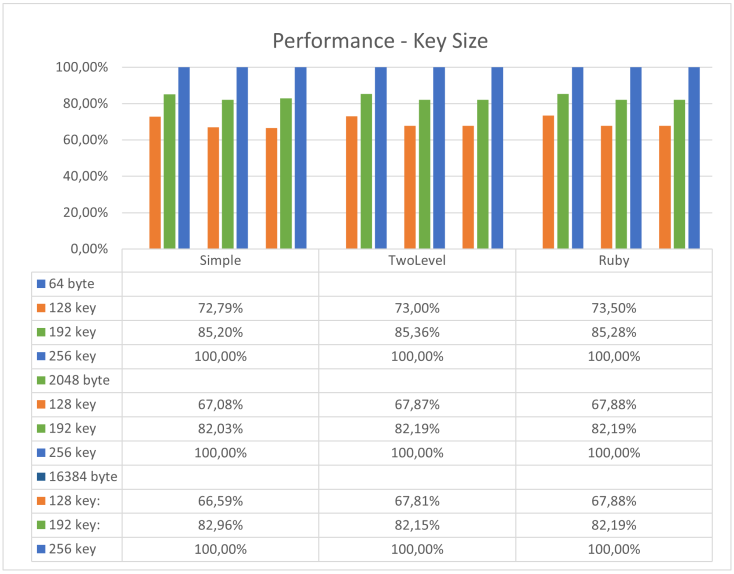
the *TempoFile* program was used to get an estimate of the net cryptographic time, however I wanted to focus my attention on a particular data that this program can provide us: in the diagram below we find the percentage of time dedicated to data encryption and decryption alone. If we want to access an encrypted file, modify it and save it again in encrypted form, a large part of the computational power should be reserved for data protection operations: in the cases studied, never less than 68% for very small files, not less than 86% in the case of non-negligible file sizes.



	128 key	192 key	256 key	2048 byte	128 key	192 key	256 key	16384 byte	128 key:	192 key:	256 key
🟦 Simple	74,84%	77,69%	80,34%		86,30%	88,51%	90,38%		88,47%	90,53%	92,02%
🟧 TwoLevel	68,25%	71,54%	74,65%		85,96%	88,11%	90,02%		86,47%	88,57%	90,41%
⬜ Ruby	68,93%	72,02%	75,12%		86,15%	88,28%	90,16%		86,88%	88,91%	90,71%

5.3 Dimensions of the Key

The AES algorithm performs 10 rounds for the 128-bit key, 12 for the 192-bit key, and 14 for the 256-bit key. Each **round** is composed of the 4 steps exposed during the intruding of the algorithm (SubBytes, ShiftRow, MixColumns, AddRoundKey). This increases the possible encryption combinations and increases their security, going from $3,4 * 10^{38}$ 128-bit key combinations up to more than $1.15 * 10^{77}$ 256-bit key combinations. All this has an increase in computational cost: the graph below shows the time of the simulations performed with a key of different length in relation to the key of maximum length. It is interesting to note that, despite the change in architecture and the size of the file used in the AES algorithm, one figure remains almost constant: the time taken to encrypt with a 128-bit key is equal to 70% of the time used with a 256-bit key instead, using a 192-bit key saves 18% of the time compared to a 256-bit key. The increase in the number of rounds is not a negligible detail!

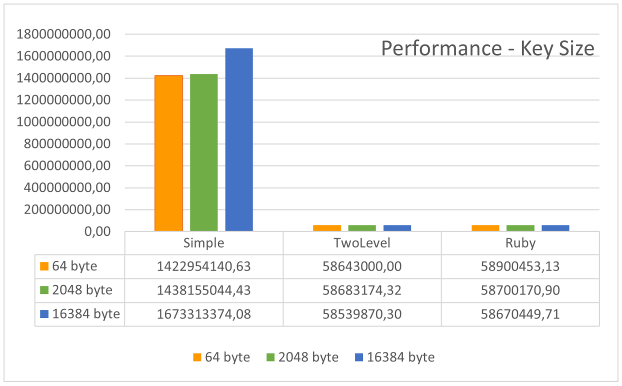


	Simple	TwoLevel	Ruby
🟦 64 byte			
🟧 128 key	72,79%	73,00%	73,50%
🟩 192 key	85,20%	85,36%	85,28%
🟦 256 key	100,00%	100,00%	100,00%
🟩 2048 byte			
🟧 128 key	67,08%	67,87%	67,88%
🟩 192 key	82,03%	82,19%	82,19%
🟦 256 key	100,00%	100,00%	100,00%
🟦 16384 byte			
🟧 128 key:	66,59%	67,81%	67,88%
🟩 192 key:	82,96%	82,15%	82,19%
🟦 256 key	100,00%	100,00%	100,00%

Figure 8: Ratio to 256-bit key encryption

5.4 File size

One thing that caught my attention is the increase in execution time with respect to changing file size. In the graph below I have reported the relationship between net execution time and file size. Observing the most powerful architectures it might seem that the execution time increases linearly with respect to the file size, however observing the "Simple" architecture this statement may not be true. The increase in the ratio in the final part is due to the fact that, since the Simple architecture is less performing, it is more stressed by changes in complexity and therefore better highlights the variation. I am inclined to think that the relationship between execution time and file size is not linear, however the data available is not sufficient for this type of analysis.

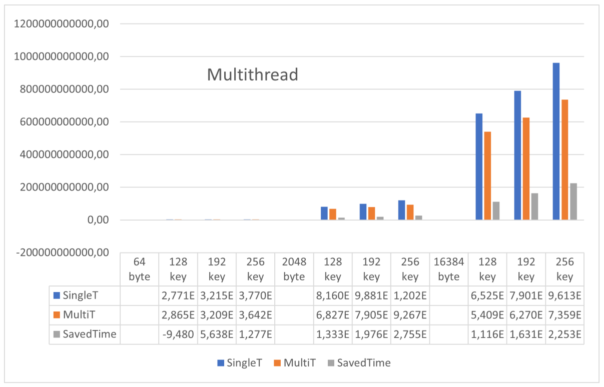


	Simple	TwoLevel	Ruby
🟧 64 byte	1422954140,63	58643000,00	58900453,13
🟩 2048 byte	1438155044,43	58683174,32	58700170,90
🟦 16384 byte	1673313374,08	58539870,30	58670449,71

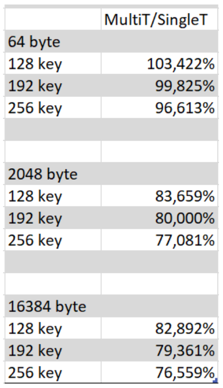
Figure 9: Time - File Size Report

5.5 Multi Thread

I focus now on the Ruby architecture, the only one that can benefit from the parallelization of operations. Comparing the 2 types of execution, whose results are shown in the graph, it is known that the time savings are lower than expected: in the fastest simulation the simulation time is even increased due to the high cost of creating threads, while in the more expensive ones it is still necessary at least 70% of the initial time.



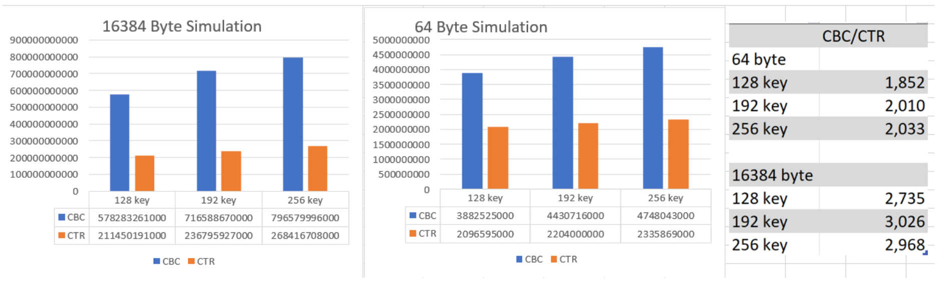
	64 byte	128 key	192 key	256 key	2048 byte	128 key	192 key	256 key	16384 byte	128 key	192 key	256 key
🟦 SingleT		2,771E	3,215E	3,770E		8,160E	9,881E	1,202E		6,525E	7,901E	9,613E
🟧 MultiT		2,865E	3,209E	3,642E		6,827E	7,905E	9,267E		5,409E	6,270E	7,359E
▨ SavedTime		-9,480	5,638E	1,277E		1,333E	1,976E	2,755E		1,116E	1,631E	2,253E



	MultiT/SingleT
64 byte	
128 key	103,422%
192 key	99,825%
256 key	96,613%
2048 byte	
128 key	83,659%
192 key	80,000%
256 key	77,081%
16384 byte	
128 key	82,892%
192 key	79,361%
256 key	76,559%

Figure 10: Single-Multi Thread

I have identified the cause of such a small increase in performance in the unfair workload balance between the CPU. So I decided to run the same simulation by disabling the threads that managed the encryption modes (CBC and CTR) one at a time obtaining the following results:



16384 Byte Simulation

	128 key	192 key	256 key
CBC	578283261000	716588670000	796579996000
CTR	211450191000	236795927000	268416708000

64 Byte Simulation

	128 key	192 key	256 key
CBC	3882525000	4430716000	4748043000
CTR	2096595000	2204000000	2335869000

	CBC/CTR
64 byte	
128 key	1,852
192 key	2,010
256 key	2,033
16384 byte	
128 key	2,735
192 key	3,026
256 key	2,968

Figure 11: CBC-CTR comparison

As expected, the bottleneck is cryptographic performance in *CBC* mode, which takes up to 3 times the time used in CTR mode to process the same file. The CBC mode cannot be further parallelized due to the use of intermediate results in its implementation, so the results obtained cannot be further improved in a consistent way.

All data and charts are available on GitHub in the "Result" folder [9]

6 Conclusion

Having reached the end of this project, I can say that I am satisfied with the objectives achieved and above all with the notions learned. With this work I had the opportunity to familiarize myself with the world of hardware simulation using specific software such as gem5, the possibility of using widely used tools that I had never used (such as cmake or latex) and above all I experienced firsthand what it means to collect data, analyze them and and arrive at some final considerations. I have personally analyzed the impact that cryptography has on the performance of the machines we use every day and I can say that I have reached a greater awareness of the subject.

Special thanks go to Prof. Christian Pilato for his support and help during the (many) difficult moments.

Bibliography

[1] 1. The gem5 simulator, 2011. URL `https://www.gem5.org/`.

[2] 2. Learning gem5, 2011. URL `http://www.gem5.org/documentation`.

[3] 3. Oracle vm virtualbox, 2007. URL `https://www.virtualbox.org/`.

[4] 4. Ubuntu, 2004. URL `https://www.ubuntu-it.org/`.

[5] 5. Compiling question, 2021. URL `https://tinyurl.com/bj67t2x6`.

[6] 6. Aes algorithm, 1998. URL `https://it.wikipedia.org/wiki/Advanced_Encryption_Standard`.

[7] 7. Block cipher, 1998. URL `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation`.

[8] 8. kokke- tiny-aes-c, 2019. URL `https://github.com/kokke/tiny-AES-c`.

[9] 9. My program, 2021. URL `https://github.com/Giuseppe-Calcagno/Progetto-IngInf`.