UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

PEDRO MARTINS BASSO

Cache Calibration for Accurate Simulation of Multi-core Systems

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engineering

Advisor: Prof. Dr. Taisy Silva Weber
Coadvisor: Prof. Dr. Paolo Rech

Porto Alegre
November 2021

ACKNOWLEDGMENTS

First of all, I would like to acknowledge Dr David Novo (LIRMM, Montpellier), Dr. Alejandro Nocua (Atos R&D Team), and Dr. Gregory Vaumourin (Atos R&D Team) for proposing me this work. They provided me a lot of knowledge and support, being always available to help me from the beginning. Also, I would like to thank the Federal University of Rio Grande do Sul (UFRGS), specially Informatics Institute, for the excellence with which you conduct everything in the educational process and for the all of the opportunities I had while studying here.

I would like to acknowledge the Prof. Dr. Paolo Rech for all the work that we developed together during my 3 years in the scientific research domain, being always an excellent advisor and being more then a professor to me, thank you for being my friend. I also would like to acknowledge Profa. Dr. Taisy Weber for accepting my invitation to be my advisor in this work .

I would like to thank my mother, my father, and my sister. I miss you all so much, thanks for always supporting me in my academic and personal decisions with love, understanding and for being always an awesome reference to me. Also, thanks to all my friends from Brazil and for all the amazing people that I met in France, I am forever grateful to have you all in my life.

Finally, I would like to thank Polytech Montpellier for giving me the unimaginable opportunity to accomplish my double degree program in France. Thank you for welcoming me so well and for making these exchange experience the best years of my life.

ABSTRACT

The computer systems market has been increasing significantly since the beginning of the cloud computing era. This demand leads to an increase on computer architectures complexity and efficiency. The simulation step is one of the most important during the development of new architectures, it eliminates the need of the real hardware during the initial developing phases. In this work, we propose an ARM Neoverse N1 gem5 simulator model. We calibrate the cache memories of the model using microbenchmarks on the model and comparing with the real hardware architecture. The results of the work show that our calibration method reaches cache delay access time accuracy close to the real hardware.

Keywords: Computer Architecture. Memory Hierarchy. Computer System Simulation. High Perfomance Computing. Cloud Computing.

Simulation of multi-core systems

RESUMO

O mercado de sistemas de computação tem aumentado significativamente desde o início da era da computação na nuvem. Esta demanda leva a um aumento na complexidade e eficiência das arquiteturas de computadores. A etapa de simulação é uma das mais importantes durante o desenvolvimento destas: ela elimina a necessidade do hardware real durante as fases iniciais do fluxo de desenvolvimento. Neste trabalho, propomos um modelo de simulação da arquitetura ARM Neoverse N1 utilizando o simulador gem5. Calibramos as memórias caches do modelo usando microbenchmarks, comparando com o hardware real que implementa esta arquitetura . Os resultados do trabalho mostram que nosso método de calibração atinge tempos de acesso às caches próximo ao hardware real.

Palavras-chave: Computer Architecture, Computer System Simulation, Memory Hierarchy, High Perfomance Computing, Cloud Computing.

LIST OF ABBREVIATIONS AND ACRONYMS

N1SPD	Neoverse N1 System Development Platform
SoC	System on a chip
IP	Intellectual Property
CAL	Component Aggregation Layer
TLB	Translation Lookaside Buffer
ITLB	Instruction Translation Lookaside Buffer
DTLB	Data Translation Lookaside Buffer
STLB	Shared Translation Lookaside Buffer
ALU	Arithmetic Logic Unit
SIMD	Single Instruction, Multiple Data
SLC	Shared Level Cache
FS	Full System Emulation
SE	System-call Emulation
MU	Memory Unit
MU	Load/Store Queue

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

1.1 Motivation and Scope

Today, the high tech landscape is shaped by the innovations of new, largescale consumer services such as 5G, Cloud Computing, Internet of Things, Big Data and Autonomous Driving. Machine Learning and Artificial Intelligence applications are fundamentally changing consumer behavior. This has in turn led to evolving platform and solution architectures to address these new applications in an efficient and scalable fashion.

The current development of the cloud computing market aims to improve performance and energy efficiency on modern computer systems. This process leads to a demand for high performance solutions, within strict latency constraints and power budgets that increase the complexity of computer architectures.

One of the first steps in computer architecture development and research is software modeling and simulation. This process allows the researchers to test new architectures without the need of the real hardware. However, one of the major factors on simulating architectures is to achieve results with good simulation accuracy and tolerable simulation time. Since its publication in 2011, the gem5 simulator is one of the most popular academic focused computer architecture simulation frameworks.

1.2 Objectives

In this work, we propose an architecture gem5 model for the ARM Neoverse N1, based on the Neoverse N1 System Development Platform (N1SPD). To develop this model, we use an ARM Cortex-A72 gem5 model to tune it to our proposed architecture (i.e, the Neoverse N1). To correlate with the real hardware, we collect architectural information from both system on a chip (SoC) and N1SDP technical documentation (ARM, 2019b; ARM, 2019a). However, some parameters are missing as they are not made public. To dribble this problem, we use microbenchmarks for model calibration (i.e., finding the correct parameter for the model instantiation) of the memory hierarchy of our gem5 model against a real hardware architecture. Our results show that with a proper calibration we can achieve average access time delays close to real hardware. This work was developped as a Master's Thesis in a BRAFITEC double degree scholarship at Polytech

Montpellier and have the collaboration of Atos R&D Product Architecture 2 Team, based at Échirolles, France, who provided access to the real platform.

1.3 Outline

The rest of the work is organized as follows: Chapter 2 presents the required background to comprehend our work, it details the ARM Neoverse N1 architecture and the gem5 simulator, Chapter 3 details our proposed methodology, presenting our gem5 architectural model and our strategies for cache level modeling, Chapter 4 presents and discuss the results that we achieved using our proposed methodology and then Chapter 5 presents the conclusions and perspectives for future research created by this study.

2 BACKGROUND

This chapter presents an overview of the concepts and technologies that were studied and used on the development of this work. Section 2.1 presents the concepts of the ARM Neoverse N1 architecture and section 2.2 presents gem5, an open source computer architecture simulator.

2.1 ARM Neoverse N1

In this work, we will be using the ARM Neoverse N1 architecture. This architecture was released in 2019 using the 7nm technology with a different focus on the market. Normally, ARM aims to the mobile market, where the company has been the leader in the last years.

The N1 architecture is the first ARM design to specifically target the infrastructure market, mainly on the cloud domain. This architecture is designed for performance as a synthesizable Intellectual Property (IP) core and is sold to other semiconductor companies to be implemented in their own chips.

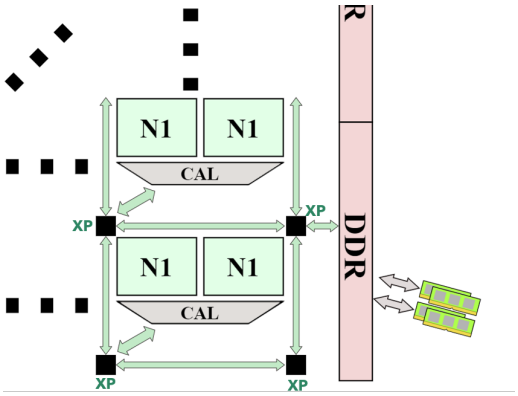
2.1.1 Architecture Overview

At high level, the Neoverse N1 architecture is based on the Cortex-A76 architecture, but the main difference is the fact that the Neoverse N1 is a core-scalable architecture, being possible to scale from 4 to 128 cores.

That means that multiples SoC's can be projected using the same architecture, but with different configurations (for different purposes) and having different performances. For networking and storage, the N1 targets 8 to 32-core designs with a thermal design power (TDP) in the range of 25 to 65 W. On the edge domain (e.g., 5G base stations), N1 targets 16 to 64 core designs targeting TDPs in the range of 35 W to 105 W. For hyperscale data centers, the N1 targets designs with 64 to as much as 128 cores with 150 W (CHIP, 2019).

Figure 2.1 shows a reduced overview on the block diagram of the architecture. There are two Neoverse N1 cores per node sitting on the same crosspoint (XP) via the component aggregation layer (CAL) which allows for two identical devices to sit on the

Figure 2.1: Reduced Neoverse N1 SoC block diagram.

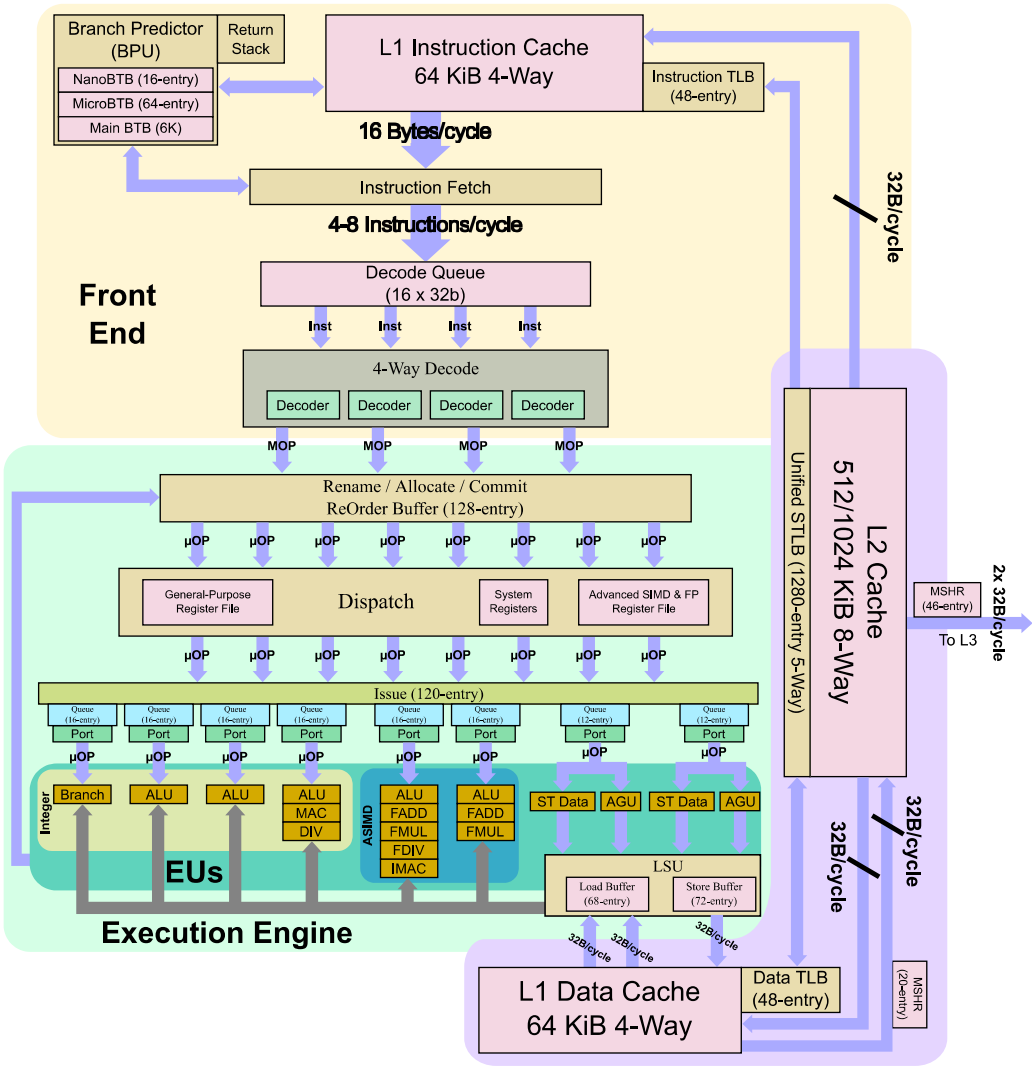


Source: adapted from (WIKICHIP, 2019)

same XP port. Neoverse N1 features 1 MB of shared system level cache (SLC) per core. Since the N1 is partitioned as duplexes, there is a 2 MB bank per duplex. Thus a 32-core design features 16 SLC banks for a total of 32 MB of cache. The largest design with 128 cores will have 64 banks for a total of 128 MB system-level cache. Neoverse N1 takes advantage of the Arm's Coherent Mesh Network 600 (CMN-600) mesh architecture to interconnect the nodes and the rest of the circuit.

Figure 2.2 shows in details an individual core of the Neoverse N1. At memory hierarchy level, the Neoverse N1 has an 64KB private 4-way set associative L1 instruction and data caches and an 8-way set associative L2 cache with a configurable size of 256KB, 512KB, or 1024KB and also the shared system level cache. It provides also a dedicated L1 Translation lookaside buffer (TLB) for instruction cache (ITLB) and another one for data cache (DTLB). Additionally, there is a unified L2 TLB (STLB). At instruction level, Neoverse N1 supports an aggressive out-of order superscalar pipeline and implements a 4-wide front-end with the capability of dispatching/committing up to eight instructions per cycle. The core deploys three ALUs, a branch execution unit, two Advanced SIMD units, and two load/store execution units. Note that L3 and SLC caches are implementation options.

Figure 2.2: Neoverse N1 individual core.



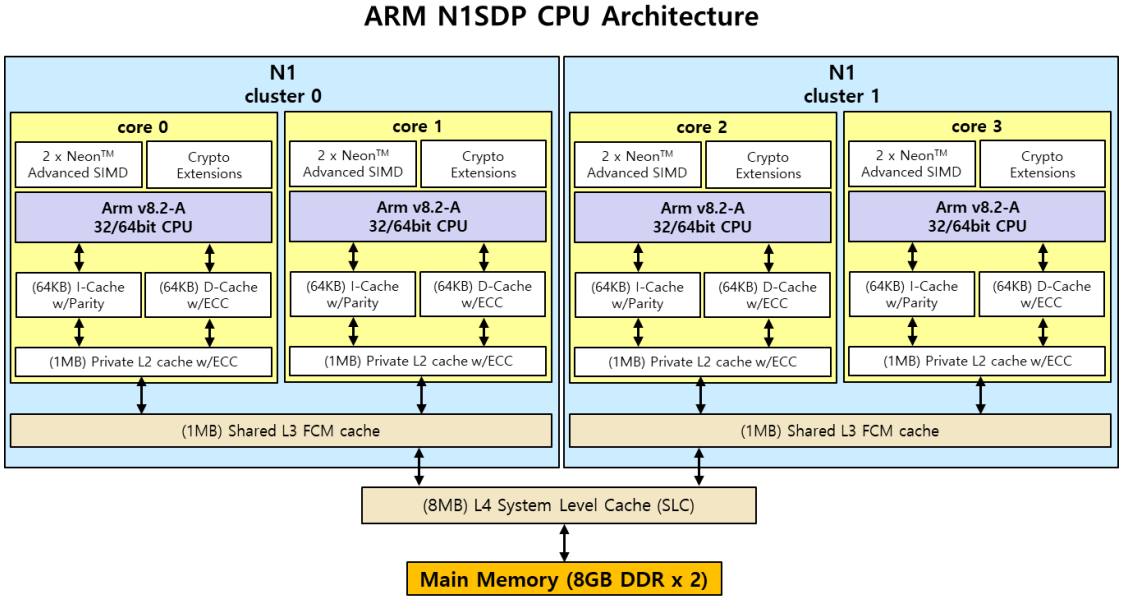
Source:(WIKICHIP, 2019)

2.1.2 The Neoverse N1 System Development Platform (N1SDP)

The Neoverse N1 System Development Platform (N1SDP) (ARM, 2019b) is an N1 CPU-based development platform for hardware prototyping, software development, system validation, and performance profiling or tuning provided by ARM. The board implements two dual-core Neoverse N1 CPU clusters (i.e. total of 4 N1 CPUs) with an operational clock speed of 2.6 GHz.

Each core on N1SPD has an L1 64KB instruction cache and data cache and a 1MB private L2 cache. Also, it has a 2MB cluster-level shared L3 and an 8MB shared L4 system level cache (SLC). The board implements the ARM CMN-600 mesh architecture and supports dual-channel DDR4-2667 memories. On this work, we will be using the N1SDP on our methodology (discussed in Section 3) to validate and compare the simulations and architectural changes done on this work.

Figure 2.3: N1SDP Architecture



Source:(HAM et al., 2021)

2.2 The gem5 Architecture Simulator

The gem5 simulator (BINKERT et al., 2011) is an open source community-supported computer architecture simulator system. Its infrastructure provides flexibility by offering

a diverse set of CPU models, system execution modes, and memory system models and it currently supports a variety of instruction set architectures (ISA) including Alpha, ARM, MIPS, Power, SPARC, and x86.

The simulator runs on Linux environment and it is developed based on Python and C++ languages. To perform a simulation the user has to instantiate the chosen architecture with the correct configuration (i.e. number of cores, caches, memory, etc.) and that can be easily done on a Python configuration script. Each architectural component (i.e. CPU, cache, memory, bus, etc.) is declared as a Python class called SimObject. SimObjects are wrapped C++ objects that are accessible from the Python configuration script. Almost all objects in gem5 inherit from the base SimObject type.

2.2.1 CPU models

The gem5 simulator provides a different set of CPU models that can be used for different purposes:

1. *In-Order Models*: Instructions executed on in-order model are statically scheduled, they are fetched, executed and completed in compiler-generated order, meaning that if a dependency occurs, all the code execution is affected by this dependency. Two main in-order CPU models are available on gem5 simulator: (1) AtomicSimpleCPU, (2) TimingSimpleCPU. The major difference between the models is that AtomicSimpleCPU uses atomic memory accesses (i.e for fast forwarding and warming up caches and return an approximate time to complete the request without any resource contention or queuing delay) while TimingSimpleCPU uses timing memory accesses (i.e most realistic timing and it includes the modeling of queuing delay and resource contention). Both CPU models implement functions to read and write memory, also the port that is used to hook up to memory, and the CPU to cache connection.
2. *Out-of-Order Models*: Instructions executed on out-of-order model are dynamically scheduled, they are fetched in compiler-generated order, but their executions can be re-scheduled to avoid dependencies and pipeline stalls (i.e. a condition that stops the pipeline execution). The O3CPU is the out-of-order model available on the gem5 simulator. It is a pipelined model that simulates dependencies between instructions, functions units, memory access and pipeline stages. In this work, we

use the O3CPU model as a base model for our methodology (detailed on section 3.1) as it is the closest to the Neoverse N1 core architecture that we try to simulate.

2.2.2 Simulation Types

The gem5 simulator provides two types of execution mode depending on the user requirements:

1. *Full System Emulation*: In Full System Emulation (FS) mode, gem5 simulates a bare-metal environment suitable for running an operating system. The FS mode is similar to running a virtual machine, it provides support to interruptions, exceptions, privilege levels, I/O devices, etc.
2. *System-call Emulation*: The System-call Emulation (SE) mode is designed to be used when the user is focused on the simulation of the CPU and the memory system, being not necessary to model the operating system. The SE mode emulates most common system calls, so whenever the program executes a system-call, gem5 traps and emulates the call, often by passing it to the host operating system. We chose to use only System-call Emulations due the fact that is not necessary to simulate the operating system and its functionalities for our proposed methodology and also due to facility reasons.

2.2.3 Simulation Example

In this section we present the steps to perform a simple simulation using the SE mode on gem5 simulator. First, it is necessary to install all of the required dependencies:

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

Then, we need to clone the official gem5 repository to our local machine using the command *git clone*:

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

At this point, we are able to compile the simulator, currently, we must compile gem5 separately for every ISA that we want to simulate, for this simple example we are going to perform a simulation using the x86 architecture. We use the *SCons* command to

compile the simulator for the chosen architecture:

```
$ scons build/X86/gem5.opt -j9
```

When compilation is finished, we should have a working gem5 executable at `<build/X86/gem5.opt>`. We are now ready to simulate in gem5. First, we introduce the corresponding command line, the gem5 command line has the following format:

```
$ <gem5_ISA_binary> [gem5_options] <simulation_script> [
  script_options]
```

This command calls the gem5 binary (compiled for the chosen ISA) and its related options, as well a simulation Python script and its corresponding options. This Python script sets up and executes simulation, by setting the SymObjects in the gem5 model, including the CPU models, caches, memory controllers, buses, etc. On this example we are going to use a python simulation script provided by the gem5 documentation and a simple *"Hello World!"* program to validate our simulation. So, we can run a simple simulation using the following command:

```
$ build/X86/gem5.opt configs/example/se.py --cmd=tests/test-progs/
  hello/bin/x86/linux/hello --cpu-type=TimingSimpleCPU --caches --
  l1d_size=64kB --l1i_size=16kB
```

Figure 2.4: Output of the simulation example.

```
basso@DESKTOP-17900ED:~/PIFE/gem5$ build/X86/gem5.opt configs/example/se.py --cmd=tests/test-progs/hello/bin/x86/linux/h
ello --cpu-type=TimingSimpleCPU --caches --l1d_size=64kB --l1i_size=16kB
warn: CheckedInt already exists in allParams. This may be caused by the Python 2.7 compatibility layer.
warn: Enum already exists in allParams. This may be caused by the Python 2.7 compatibility layer.
warn: ScopedEnum already exists in allParams. This may be caused by the Python 2.7 compatibility layer.
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 20.0.0.3
gem5 compiled Sep 20 2020 20:46:31
gem5 started Jan 28 2021 21:58:55
gem5 executing on DESKTOP-17900ED, pid 1025
command line: build/X86/gem5.opt configs/example/se.py --cmd=tests/test-progs/hello/bin/x86/linux/hello --cpu-type=Timin
gSimpleCPU --caches --l1d_size=64kB --l1i_size=16kB

Global frequency set at 1000000000000 ticks per second
warn: failed to generate dot output from m5out/config.dot
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7000
**** REAL SIMULATION ****
info: Entering event queue @ 0.  Starting simulation...
Hello world!
Exiting @ tick 31680000 because exiting with last active thread context
```

Source: the authors

Here, *"se.py"* is the simulation script and we pass as positional arguments: (1) *"hello"*, the program binary file, (2) *"cpu-type"*, the CPU model (TimingSimpleCPU for this example) and (3) *"l1d_size"* and *"l1i_size"*, the size specifications (64KB for this example) for the L1 data cache and L1 instruction cache, respectively.

Figure 2.4 shows the terminal output for our simulation example, we can observe that we have some warnings that can be safely ignored at this moment. Then, at the end of

the output we can see that the simulator was able to execute our *"Hello World!"* program. On Section 2.2.4 we will be presenting all the generated outputs and statistics provided by gem5 simulator.

2.2.4 The gem5 outputs and statistics

The gem5 simulator provides output files and statistics after the simulation run for data studies and debugging. Normally, there are three files generated in a directory chosen by the user: *"config.ini"*, *"config.json"* and *"stats.txt"*. The *"config.ini"* file contains a list of every SimObject (CPU models, caches, memory controllers, buses, etc.) created for the simulation and the values for its parameters. The *"config.json"* file is the same as *"config.ini"*, but in *json* format.

The *"stats.txt"* file collects all statistics regarding the simulation, as Figure 2.5 indicates. Each instantiation of a SimObject has its own statistics, so we have statistics for the CPU (number of float instructions executed, for example), the memories (total read bandwidth, for example) and the simulation itself (number of instructions simulated, for example).

Figure 2.5: Fragmented example of a *"stats.txt"* file.

```
---------- Begin Simulation Statistics ----------
final_tick                                   22155750                       # Number of ticks from beginning of simulation
host_inst_rate                                  97427                       # Simulator instruction rate (inst/s)
host_mem_usage                                2434000                       # Number of bytes of host memory used
host_op_rate                                   112763                       # Simulator op (including micro ops) rate (op/s)
host_seconds                                     0.05                       # Real time elapsed on the host
host_tick_rate                               428093722                       # Simulator tick rate (ticks/s)
sim_freq                                  1000000000000                       # Frequency of simulated ticks
sim_insts                                        5028                       # Number of instructions simulated
sim_ops                                          5834                       # Number of ops (including micro ops) simulated
sim_seconds                                  0.000022                       # Number of seconds simulated
sim_ticks                                    22155750                       # Number of ticks simulated
system.cpu_cluster.cpus.committedInsts           5028                       # Number of instructions committed
system.cpu_cluster.cpus.committedOps             5834                       # Number of ops (including micro ops) committed
system.cpu_cluster.cpus.cpi                 17.625895                       # CPI: cycles per instruction
system.cpu_cluster.cpus.discardedOps             1297                       # Number of ops (including micro ops) which were
system.cpu_cluster.cpus.idleCycles              78327                       # Total number of cycles that the object has spe
system.cpu_cluster.cpus.ipc                  0.056735                       # IPC: instructions per cycle
```

Source: the authors

3 EVALUATION METHODOLOGY

On this section we present the methodology used for this work. Section 3.1 presents the development of the Neoverse N1 gem5 model and Section 3.2 presents an analysis at system cache level to calibrate our gem5 model.

3.1 Neoverse N1 gem5 model

At the beginning of this work, we have started performing several simulations using different available CPU models and architectures to get familiar with the simulator. Then, we developed a gem5 CPU model for the Neoverse N1 architecture. The model is based on an ARM Cortex-A72 CPU (ARM, 2014) gem5 model provided by Atos R&D. This CPU model implements also an out-of-order architecture.

Figure 3.1: Neoverse N1 gem5 model Python Class.

```
class ARM_Neoverse_N1(DerivO3CPU):
    LQEntries = 48
    SQEntries = 48
    LSQDepCheckShift = 0
    LFSTSize = 1024
    SSITSize = 1024
    decodeToFetchDelay = 1
    renameToFetchDelay = 1
    iewToFetchDelay = 1
    commitToFetchDelay = 1
    renameToDecodeDelay = 1
    iewToDecodeDelay = 1
    commitToDecodeDelay = 1
    iewToRenameDelay = 1
    commitToRenameDelay = 1
    commitToIEWDelay = 1
    fetchWidth = 3
    fetchBufferSize = 16
    fetchToDecodeDelay = 1
    decodeWidth = 3
    decodeToRenameDelay = 1
    renameWidth = 3
    renameToIEWDelay = 1
    issueToExecuteDelay = 1
    dispatchWidth = 5
    issueWidth = 8
    wbWidth = 8
    fuPool = ARM_Neoverse_N1_FUP()
    iewToCommitDelay = 1
    renameToROBDelay = 1
    commitWidth = 8
    squashWidth = 8
    trapLatency = 13
    backComSize = 5
    forwardComSize = 5
    numIQEntries = 92
    numROBEntries = 192

    switched_out = False
    # branchPred = ARM_Neoverse_N1_BP()
    branchPred = Param.BranchPredictor(TournamentBP(
        numThreads = Parent.numThreads), "Branch Predictor")
```

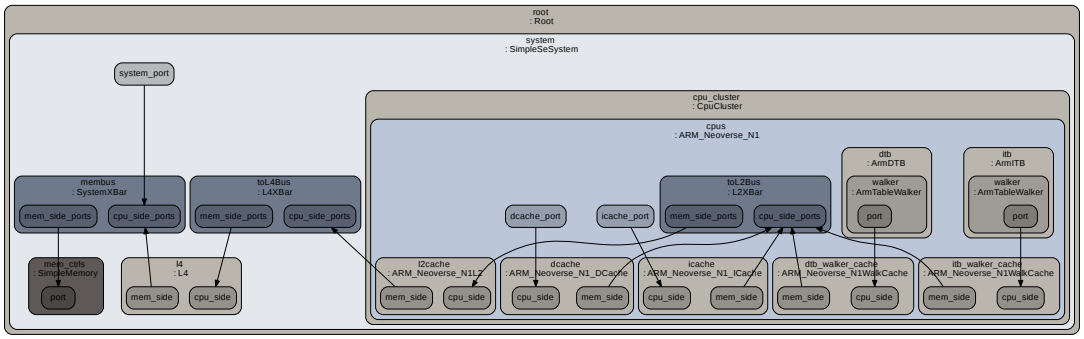
Source: the authors

To have all the out-of-order functionalities, we derived our Neoverse N1 model from the O3CPU model Python class. Figure3.1 shows the Python class that describes some main parameters from the core of the architecture. In the same file, we have also

instantiated the functional units, the branch predictor, ALU instructions, floating point instructions and also the cache memories (i.e. instruction and data L1, TLB's, L2 and L4).

Figure 3.2 shows the block diagram generated by gem5 for our Neoverse N1 model. We can observe that it has one N1 CPU in the CPU cluster, with its instruction and data L1 caches and TLB's, L2 cache, also all the buses that interconnects each module. Out of the CPU cluster, we have the main memory (represented as SimpleMemory) and the L4 cache. We have instantiated only a single N1 CPU core to be easier to study and validate our methodology. This choice in using only a single CPU core affects the cache L3 usage, due the fact that this cache level is used for communication between the two CPUs sharing a cluster. Therefore, we do not instantiate the L3 cache in our simplified gem5 model.

Figure 3.2: Block diagram of the Neoverse N1 model provided by gem5.



Source: the authors

In this work, we focus on calibrating the caches of the model to be as close as possible with the real SoC hardware (i.e, the N1SDP board). At the beginning, we have used a generic gem5 cache module (with default values for size, associativity and data latency) to instantiate the caches of the architecture and test the functionality of the model. Then, we used the Neoverse N1 (ARM, 2019a) and N1SDP (ARM, 2019b) technical documentations as the main sources of information to model the architecture. This type of documentation provides general information about the architecture that are helpful for this process. We can find details about each cache level (e.g, caches sizes, associativity, etc.) and the CPU itself (e.g, pipeline functionality, clock frequency, etc.) that are important for our model. However, the translation from documentation to simulator parameters is not simple and some key parameters are not public (e.g, the cache access times, the number

Table 3.1: List of default and calibrated parameters for the Neoverse N1 model extracted from technical documentation.

gem5 Cache module	gem5 cache Parameter	Default	Calibrated
L1_ICache	size	32 KB	64 KB
	replacement_policy	LRURP	LRURP
	writeback_clean	false	true
	associativy	2	4
L1_DCache	size	32 KB	64 KB
	replacement_policy	LRURP	LRURP
	writeback_clean	false	true
	associativy	2	4
L2_Cache	size	1MB	1MB
	associativy	16	8
L4_Cache	size	N/A	8MB

Source: the authors

of parallel access that can be processed by a cache, or the memory controller buffering latency).

3.2 Cache level analysis

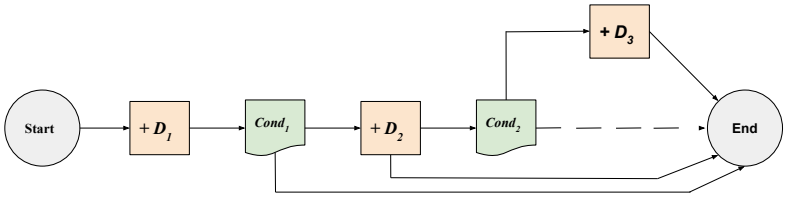
Initially, the caches of the model were instantiated with default parameters provided by the generic gem5 cache module. Then, we started calibrating the parameters that are easily provided by the documentation. Table 3.1 shows the default and calibrated parameters that were found in the technical documentation. In this part of the work, we calibrated the caches size and associativy. Note that the L4 cache is not instantiated by default on the gem5 caches modules, so we instantiated it with the correct values. We simulate, then, the model with the default and calibrate parameters and we conclude that the default model provides results that are not close to our architecture. However, even if the calibrated model provides good results, we decide to extract the caches parameters that are not listed in the architecture technical documentation.

We target, then, to find the correct data latency on each cache level (i.e, the required number of cycles to access each cache level). To identify the simulator parameters related with the memory hierarchy, we study the path that a memory request follows during simulation.

When a memory instruction is executed by the *Memory Unit* (MU), the instruction is issued to a *Load/Store Queue (LSQ)* and a memory request is generated. The instruction

stays in the LSQ until the memory request is fully executed. When a memory request is issued, the data goes through the different levels of the memory hierarchy depending on where the data sits. If a request misses all cache levels, it goes to the main memory (represented as *SimpleMemory* in our model).

Figure 3.3: The delay of a memory request increases as it deepens in the hierarchy. Access time begins in start until end.



Source: the authors

The memory request departs from the MU of the CPU core and accumulates delay as it moves deeper in the memory hierarchy. If we sum all of these individual delays we find the request access time, as illustrated in Figure 3.3. We selected, then, three cache parameters that are related with cache latency: *data_latency*, *tag_latency* and *response_latency*. All of them control the required number of cycles to access and move data at each cache level.

To discover these parameters, we execute microbenchmarks on the real hardware (i.e, the N1SDP board). A microbenchmark is a small program that was developed to extract these missing cache parameters. We exploit the memory dependencies of the architecture to perform our microbenchmark with the following structure:

1. *Data Pinning*: The microbenchmark initializes data in a targeted level of the memory hierarchy and uses a fixed access pattern to force all memory requests to follow a predetermined path of the delay model. The data can be pinned in different cache levels and in the main memory. We set the location of the data by controlling the size of an array we repeatedly access. For example, if we target the L2 data latency, we should set an array size that fits in L2 and, consequently, the cache L1 miss rate should be close to 100%.
2. *Memory request dependency*: The program avoids or trigger conflicts that may occur when the program is performing multiples access on the memory hierarchy. These conflicts can modify the path of the delay model (e.g, the L2 cache bank is busy with a previous request).

3. *Access time measurement*: The program iterates through the memory access pattern many times in a loop, such that the overall execution time interval is large enough to average out transient effects. Then, the body loop is unrolled multiple times to minimize the impact of the iterator count and the conditional check instructions.

Figure 3.4: Code example for microbenchmark.

```
1  int main(){
2  long long int array[N];
3  long long int* ptr_1 = NULL;
4
5  /* Make the chasing pointer array */
6  srand(time(0));
7
8  //Init n variable
9  n = (1<< MIN);
10 incre = n >> GRANU;
11
12     //Main loop
13     while(n < (1<<MAX)){
14
15         check = chasing(array, n); // Random linked caches
16         ptr_1 = &array[0]; // Initiate pointer to a cache line
17
18         tot_n = n;
19
20         if(check == ERR_CHASING)
21             {ERR_CHASING_MSG}
22
23         acc = 0;
24
25         start = clock(); // Start mesuring time
26
27         /* Measure LOOP */
28         for(i = 0; i < MAX/2; i++){ // Region of Interest
29             ptr_1 = *ptr_1;
30         }
31
32         end = clock();  // Stop mesuring time
33
34         cpu_time_used = ((double)(end- start))/ CLOCKS_PER_SEC;
35
36         //To be sure the array have been read
37         if( ptr_1 == NULL ){
38             printf("-- Error *end--\n");
39         }
40
41         printf("Size, Time ,Average Time Access(ns) \n");
42
43         printf("%lld,%lld,%f,%f\n",
44                 (tot_n >> S),
45                 cpu_time_used,
46                 (float)(cpu_time_used*1000000000/acc) );
47
48
49         /* Incrementation in function of the granularity process */
50         n += incre;
51         if( n == (incre << (GRANU + 1))){
52             incre = n >> GRANU;
53         }
54
55     }
56 return 0;
57 }
```

Source: the authors

Figure 3.4 illustrates a quick code overview of the microbenchmark adapted from (HUPPERT et al., 2021). The program generates *MAX* sequential load requests to a contiguous region of memory named *array* of size *N*. In line 16, we initialize each element in array allocated to a first word of a cache line. The initialized elements include the address

of another randomly chosen element also allocated to a first word of a cache line. The program closes a circular chain of references not repeating any address. Therefore, we can generate an unlimited sequence of random read requests to different cache lines by iterating over array with a pointer (i.e., pointer chasing) as shown in line 29. Accordingly, we select *MAX* to be several orders of magnitude larger than N. We can, then, control the size of array and the target cache area with *N*.

To avoid cache line locality, the program only accesses a single word of each cache line in array. We randomize the sequence to avoid triggering data prefetching that may interfere on the computation of the cache delay access. Also, the microbenchmark should traverse the array many times for cold start misses and work correctly.

For example, if the array is smaller than L1 data cache (L1D), then the array resides in the L1D size area, thus the whole execution and every load request takes the access time of L1D. Then, we can collect the L1D access time by dividing the time spent in the region of interest by MAX (i.e., the number of load requests). Alternatively, if array is much larger than L1D but still smaller than L2, then the accesses to array always miss L1D and hit L2, which allows us to measure L2 access time. The same process can be repeated with an even larger N to measure higher levels of the memory hierarchy. We present, then, in the next chapter, the performed simulations using our proposed microbenchmark and the achieved results.

4 RESULTS

This chapter presents the simulations and data extracted from the real hardware and the gem5 model using our proposed methodology.

4.1 Architecture simulations and Model Calibration

4.1.1 Neoverse N1 gem5 Model Checking

Figure 4.1: Simple benchmark for first validation of the Neoverse N1 gem5 model.

```
5   #define SIZE 20
6   static double arr[SIZE];
7
8   void add_vector(){
9
10      for (int i=0; i < SIZE; ++i) {
11
12          arr[i] = i;
13      }
14  }
15
16  int main()
17  {
18      printf(" Calling add vector function \n");
19      add_vector();
20
21      return 0;
22  }
```

Source: the authors

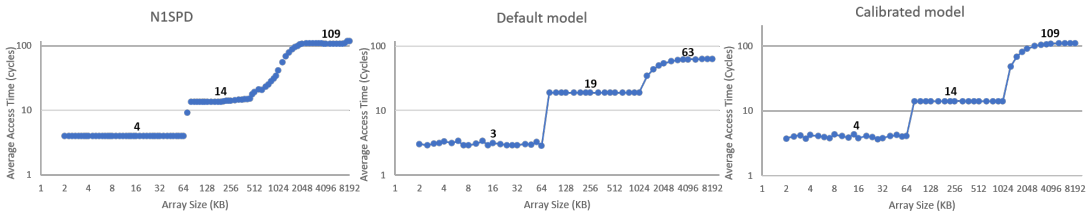
One of the first step after developing CPU models on gem5 is to be sure that what we want to simulate matches with our described model. We perform, then, a simple simulation using a simple benchmark, presented in Figure 4.1, that executes a vector aggregation on a global variable in our Neoverse N1 gem5 model. At this point of the simulation process, we can generate the block diagram described on Figure 3.2 to check if every component on the model is well instantiated and if the architecture hierarchy is respected. We check also the output files provided by the simulator (i.e, *stats.txt* and *config.ini* files) to confirm that the parameters of each component (e.g, cache and memory sizes) and all simulation settings (e.g, CPU frequency, main memory type) match with the ones that we instantiate in our gem5 model and also in the Python configuration file. This procedure helps the debugging process due the fact that this simple benchmark has a short simulation time, so it is easy to check and validate each modification that we perform on our Neoverse N1 gem5 model.

4.1.2 Model calibration using microbenchmark

After validating and debugging our gem5 model, we pass to the calibration step. At this point, we have to extract data from the real hardware to find the missing cache latency parameters. We execute, then, the microbenchmark described on Section 3.2 in the N1SPD platform. We target to find the access times for caches L1, L2 and L4, so we need to choose an array size N that reaches all the cache levels. Note that our implementation does not instantiate L3 due the fact that we are simulating a single N1 core and the L3 cache works only with dual-core clusters.

Originally, the microbenchmark provides the access time in nanoseconds. To convert it to clock cycles we use the real hardware clock period. The N1SDP operates at a frequency of 2.6GHz, so we extract its clock period (i.e, $\approx$ 0.385 ns) from it and then we calculate the average access time in CPU clock cycles by dividing the access time in nanoseconds by the clock period.

Figure 4.2: Microbenchmark running on the real hardware and on the gem5 simulator with default and calibrated configurations, respectively.



Source: the authors

The graph on the left of Figure 4.2 shows the average access time, in cycles, measured with the microbenchmark on N1SDP for different sizes of array. We note that, at the cache L1 level (i.e, the range of 2KB to 64KB) the real hardware needs 4 cycles to access data at L1 level and, for the L2 cache (i.e, the range of 64KB to 1MB), the N1SDP takes 14 cycles to access data. Then, for L4 (i.e, the range of 1MB to 4MB) the board takes 109 cycles.

Once we have the average number of cycles of the real hardware, we can pass to the calibration phase. At this point, we have to find and tune the cache parameters that are related with data latency in our gem5 model: *data_latency*, *tag_latency* and *response_latency*. *Data_latency* controls the cache data access latency, *tag_latency* controls

the tag lookup latency and *response_latency* controls the latency for the return path when a cache miss occurs. All three parameters are instantiated in number of clock cycles.

Figure 4.3: L1 data cache instantiation code with modified latency parameters.

```
# Data Cache
class ARM_Neoverse_N1_DCache(Cache):
    tag_latency = 3  #  calibrated 3   // default 2
    data_latency =  3  #   calibrated 3 // default 2
    response_latency = 3 # calibrated 3 // default 2
    mshrs = 6
    tgts_per_mshr = 8
    size = '64kB'
    assoc = 4
    write_buffers = 16
    # Consider the L2 a victim cache also for clean lines
    writeback_clean = True
```

Source: the authors

During the process of calibration, we have to change these values to reach as close as possible to the real hardware. However, we observed that if we set the parameters with the same values that we found running the program on the N1SDP, the model will not correspond with the real architecture. We explain this behavior by the fact that the gem5 simulator adds simulation delay cycles at each cache level affecting the overall access time. Therefore, we calibrate the model by analyzing at first the average access time in nanoseconds on the simulation and correlating it with the values extracted in N1SPD.

Figure 4.4: Proposed simulation script for cache modeling.

```
#!/bin/bash

WORKING_DIR="/auto/pmartinsbasso/PIFE/gem5/"
BENCH_DIR="/auto/pmartinsbasso/PIFE/benchmarks/"

${WORKING_DIR}/build/ARM/gem5.opt --listener-mode=off --outdir=/auto/pmartinsbasso/PIFE/m5out/output_modeling_N1/ \
${WORKING_DIR}/configs/example/arm/se.py \
--mem-type=SimpleMemory --mem-size=16GB --mem-channels=1 \
--cpu=neoverseN1 \
--cpu-freq=2.6GHz \
--last-cache-level=4 \
${BENCH_DIR}/modeling_microbenchmark
```

Source: the authors

Figure 4.4 shows an overview of the simulation script used to perform our simulation with our proposed microbenchmark. Table 4.1.2 lists the default and calibrated cache latency parameters for our Neoverse N1 gem5 model. The value of each parameter corresponds to the average access time, in cycles, for the each gem5 cache module. Figure 4.3 shows the L1 data cache instantiation in gem5 where we modified the latency parameters.

In this process, we start by calibrating the L1 data cache. In order to facilitate the calibration and decrease simulation time we adapted the microbenchmark to reach only

a specific cache size and not a range of sizes. For L1D, we run the program focusing in the middle of the L1D memory area by setting the program to perform the timing measure at 64KB. Then, we compare the average delay access time with the real hardware. By changing the three cache parameters we reach to values that generate timing values that correspond with the N1SDP ones. We repeat, then, the same calibration process by changing the specific cache size for 256KB and 4MB for caches L2 and L4, respectively.

Table 4.1: List of default and calibrated parameters extracted from microbenchmark.

gem5 Cache module	gem5 cache Parameter	Default	Calibrated
L1_DCache	tag_latency	2	3
	data_latency	2	3
	response_latency	2	3
L2_Cache	tag_latency	12	5
	data_latency	12	5
	response_latency	12	5
L4_Cache	tag_latency	30	89
	data_latency	30	89
	response_latency	30	89

Source: the authors

For reference, the graph on the center Figure 4.2 shows the average access times of running our microbenchmark on the gem5 model using the default latency parameters. We observe that the L1 and L4 access times are lower than the ones of the real hardware, while the opposite is true for the L2 access time. Then, the graph on the right side of Figure 4.2 shows the average access time measured with our simulation model after modeling all cache levels. We note that the new average access times of the calibrated model are practically identical to those of the real hardware. We can observe that for both simulation model (i.e default and calibrated) we have some small oscillations in the average access time values due the fact that gem5 simulator adds simulation delay cycles as explained before, but even with this effect, we can demonstrate that by a proper parameter calibration, a gem5 architecture simulation can become quite accurate.

5 CONCLUSION

In this work, we propose a method for modeling the ARM Neoverse N1 architecture in gem5 simulator. Also, we present a method for instantiating and calibrating the key timing parameters of the cache hierarchy of the model using microbenchmarks and correlating with the real ARM N1SPD SoC. Our results show that with our calibrating proposed method, we can reach average access time closes to the real hardware, without losing accuracy and with fair simulation time.

In future work, we plan to continue modeling our gem5 model by increasing the number of CPUs and modeling the L3 cache and the main memory. The main memory can be better explored by instantiating a dual channel DDR4 instead of a SimpleMemory. Also, we can run benchmark suites on the model to compare the performance (e.g, execution time, instructions per cycle, etc) of our calibrated model and the real hardware.

REFERENCES

ARM. **ARM Cortex-A72 MPCore Processor Technical Reference Manual**. 2014. <https://developer.arm.com/documentation/100095/0003>.

ARM. **Arm Neoverse N1 Core Technical Reference Manual**. 2019. <https://developer.arm.com/documentation/100616/0301>.

ARM. **Arm Neoverse N1 System Development Platform Technical Reference Manual**. 2019. <https://developer.arm.com/documentation/101489/0000/>.

BINKERT, N. et al. The gem5 simulator. **SIGARCH Comput. Archit. News**, 2011.

CHIP, W. **Arm Launches New Neoverse N1 and E1 Server Cores**. 2019. <https://fuse.wikichip.org/news/2075/arm-launches-new-neoverse-n1-and-e1-server-cores/2/>.

HAM, J.-S. et al. HPC LINPACK Parameter Optimization on Homo-/Heterogeneous System of ARM Neoverse N1SDP. In: **The International Conference on High Performance Computing in Asia-Pacific Region**. [S.l.: s.n.], 2021. p. 139–143.

HUPPERT, Q. et al. Memory hierarchy calibration based on real hardware in-order cores for accurate simulation. In: **Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.: s.n.], 2021.

WIKICHIP. **Neoverse N1 Microarchitecture**. 2019. <https://en.wikichip.org/wiki/arm_holdings/microarchitectures/neoverse_n1>.