Politecnico di Milano

HIGH PERFORMANCE PROCESSORS AND SYSTEMS (UIC 569)

RISC CORE SPECIALIZATION FOR MD

Professor **Marco Domenico Santambrogio**

Supervisor **Francesco Peverelli**

Student **Emilio Ingenito – 10863016**

Academic Year 2021 – 2022

Abstract

Aim of this project was to analyze and optimize a portion of code belonging to *miniMD*, a micro-application for molecular dynamics simulation. In particular, we decided to extrapolate the function called *compute_original*, which deals with the computation of all the forces among each atom and its neighbors. Once correctly isolated the function, we exploited the *chipyard* framework to simulate, test and benchmark it. This work was was carried out through **Verilator**, an open source RTL simulator, which enabled us to test and simulate our piece of code with different RTL generators, such as Rocket Chip. Taking advantage of the possible customization that this framework allows, we modified its components: the main customization was made with regard to the core, as we simulated Rocket Chip with Rocket Core and the Berkeley Out-Of-Order Machine(BOOM). By analyzing and accurately modifying the parameters of the simulation, we were able to reduce the clock cycles needed to execute the core functionality of our function by more than 75%.

Contents

Chapter 1

Introduction

This first chapter will provide the reader with a brief introduction to the following topics:

- introduction to the analysis of molecular dynamics, and its impact on biology;
- introduction to miniMD, the molecular dynamics simulator containing the function we decided to isolate and analyze;
- introduction to Chipyard, the framework we choose to carry out the simulations.

1.1 The context: Molecular Dynamics Simulation

The study of the macromolecular structure is a key point in the understanding of biology. Biological function is based on **molecular interactions**, and these are a consequence of macromolecular structures. Since initial structure determinations in the 50s, both in the protein and in the nucleic acid worlds, the increase in the knowledge of how macromolecular structures are built has been continuous.

In particular, the study and prediction of different types of interactions between groups of proteins is one of the growing fields in modern systems biology. On a more practical note, protein three-dimensional (3D) structures are the basis for structure-based drug design. Furthermore, both protein and nucleic acids are flexible entities, and **dynamics** can play a key role in their functionality. Proteins undergo significant conformational changes while performing their function: in general, each task they accomplish implies some structural rearrangement. This aspect makes crucial to deeply understand and study the dynamics between the molecules which make them up, in order to be able to predict their future behaviour. The figure above shows a superimposition of experimental *acetylcholinesterase* structures. There are no changes in the overall fold, just small rearrangements in the structure; however, these differences are large enough to fool ligand-docking algorithms. Larger **conformational changes** are also present in the known protein structures. Additionally, some features of protein function can be understood only when dynamic properties are taken into account. For instance, diffusion of small substrates through heme-dependent enzyme molecules requires the transient appearance of channels in the protein structure. Also, cavities have transient phenomena that in some cases can only be revealed or analyzed following its dynamics.

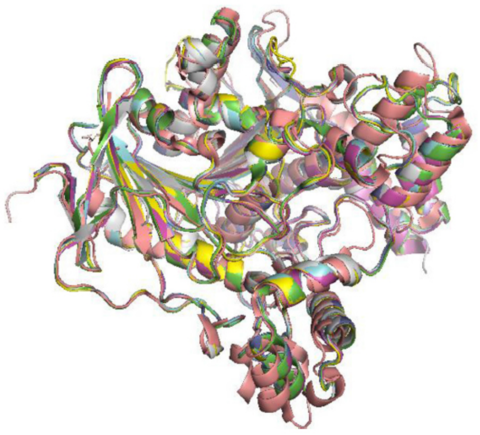

Figure 1.1. Structure variability within a protein family.

1.1.1 MD simulation

Molecular dynamics (MD) simulation, first developed in the late 70s, has advanced from simulating several hundreds of atoms to systems with biological relevance, including entire proteins in solution with explicit solvent representations, membrane embedded proteins, or large macromolecular complexes like nucleosomes or ribosomes. Simulation of systems having 50,000–100,000 atoms are now routine, and simulations of approximately 500,000 atoms are common when the appropriate computer facilities are available. This remarkable improvement is in large part a consequence of the use of high performance computing, and the simplicity of the basic MD algorithm (1.2).

An initial model of the system is obtained from either experimental structures or comparative modeling data. The simulated system could be represented at different levels of detail. **Atomistic representation** is the one that leads to the best reproduction of the actual systems. Solvent representation is a key issue in system definition. Several approaches have been followed, but the most effective is the simplest one, the explicit representation of solvent molecules, although at the expense of increasing the size of the simulated systems. Explicit solvent is able to recover most of the solvation effects of real solvent including those from entropic origin like the hydrophobic effect.

1.1.2 Basic algorithm steps

Once the system is built, some critical computations are carried on, such as the ones involving the forces acting on every atom and their potential energy, which is deduced from the molecular structure. Force-fields currently used in atomistic molecular simulations differ in the way they are parameterized. Parameters are not necessarily interchangeable, and not all force-fields allow to represent all molecule types. Once the forces acting on individual atoms are obtained, classical Newton's law of motion is used to calculate accelerations and velocities and to update the atom positions. As integration of movement is done numerically, to avoid instability, a time step shorter than the fastest movements in the molecule should be used. Algorithmic advances, that include fine-tuning of energy calculations, parallelization, or the use of graphical processing units (GPUs), have largely improved the performance of MD simulations.

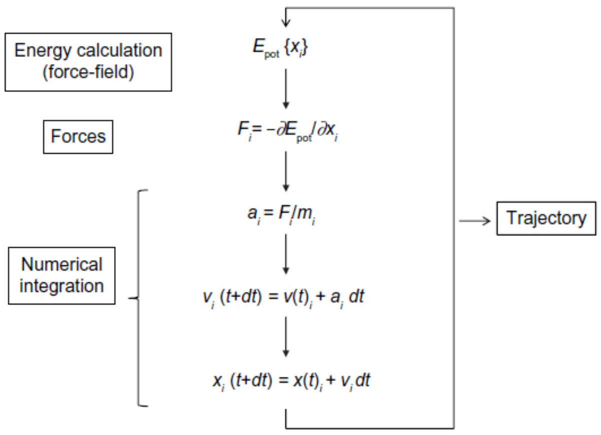
Energy calculation (force-field)
Forces
Numerical integration
$E_{pot}\{x_i\}$
$F_i = -\partial E_{pot}/\partial x_i$
$a_i = F_i/m_i$
$v_i\,(t+dt) = v(t)_i + a_i\,dt$
$x_i\,(t+dt) = x(t)_i + v_i\,dt$
Trajectory

Figure 1.2. One crucial step of the algorithm: compute forces

1.1.3 Later improvements

The present generation of computers takes benefit of parallelism and accelerators to speed-up the process. To benefit the locality of interactions, the general strategy is to distribute the system to simulate among processors. This strategy is called **spatial decomposition**: this allows to simulate, in each processor, just a small fragment of the whole system. The most efficient division is not based on the list of particles, but on their position in space. Each processor deals with a region of space irrespective of the amount and the type of the particles which are present there. Communication between processors is also reduced, as only those simulating neighboring regions have to share information. As stated, the use of accelerators, mainly GPU, has become a major breakthrough in simulation codes. Originally designed to handle computer graphics, GPUs have evolved into general-purpose, fully programmable, high-performance processors and represent a major technical improvement to perform atomistic MD. Most major MD codes have already been prepared for GPUs, and even MD codes written specifically to be used on GPUs have been developed (ACEMD61).

1.2 A microapp for MD simulation: miniMD

miniMD is a parallel molecular dynamics simulation package written in C++ and intended for use on parallel supercomputers and new architectures for testing purposes. The software package is meant to be simple, lightweight, easily adapted to new hardware and to be very scalable: any reasonable parallel computer should be able to achieve excellent scaled speedup. Furthermore, it is a simulation package which performs parallel molecular dynamics simulation of a Lennard-Jones or a EAM system and gives timing information.
The project includes many different variants of miniMD:

- **miniMD-ref** : supports MPI and OpenMP hybrid mode;
- **miniMD-OpenCL** : an OpenCL version of miniMD, uses MPI to parallelize over multiple devices;

- **miniMD-Kokkos** : supports MPI and uses Kokkos on top of it, compiles with pThreads, OpenMP or CUDA backend;
- **miniMD-KokkosLambda** : another Kokkos variant making extensive use of C++11;
- **miniMD-Intel** : supports MPI and OpenMP hybrid mode. Optimized by Intel. Comes with an intrinsic version of the LJ-force kernel and the neighborlist construction for Xeon Phi;
- **miniMD-OpenACC**: supports MPI and OpenACC hybrid mode.

Each variant is self contained and does not reference any source files of the other variants.

1.2.1 Strengths and Weaknesses

miniMD consists of less than 5,000 lines of C++ code. Like LAMMPS (a classical molecular dynamics code with a focus on materials modeling), miniMD uses spatial decomposition MD, where individual processors in a cluster own subsets of the simulation box. Furthermore, it enables users to specify:

- problem size
- atom density
- temperature
- timestep size
- number of timesteps
- particle interaction cutoff distance

Compared to LAMMPS, MiniMD's feature set is **extremely limited**, and only two types interactions (Lennard-Jones/ EAM) are available. No long-range electrostatics or molecular force field features are available. Inclusion of such features is unnecessary for testing basic MD and would have made miniMD much bigger, more complicated, and harder to port to novel hardware. The biggest difference to LAMMPS, in terms of performance, is caused by using only a single atom-type. Thus all force parameter lookups are simple variable references, while in LAMMPS they are gather operations.

MiniMD uses "neighborlists" for the force calculation, as opposed to cell lists which are employed by for example COMD (a methodology to generate all atomic transition pathways between any two structures/substates of a protein). The former approach (or variants of it) is used by most commonly used MD applications, such as LAMMPS, Amber and NAMD, while the latter is employed by some specialised codes, in particular for very large scale simulations which might be memory capacity limited. With neighborlists the memory footprint of a simulation is significantly larger, though with about 500,000 atoms per GB it is still small compared to many other applications. On the other hand the number of distance checks in the neighborlist approach is much smaller than with cell lists.

In the latest versions of miniMD, it is now simulated the behaviour of having multiple atom types. Mostly this means that certain variable accesses, such as force parameters and cutoffs, are now replaced by table lookups: this will, obviously, reduce performance compared to previous variants of miniMD. On the upside this change closes the biggest gap between the miniApp and what happens in real apps.

1.2.2 Simulation example

Let's analyze a possible example of simulation: we should first set some environment parameters to perform the desired simulation (e.g. the type of force, the number of atoms, the number of neighbors etc.) and then choose how to run the simulation (e.g. number of cores etc.). The results are listed below:

```
Run Settings:
    # MPI processes: 2
    # OpenMP threads: 16
    # Inputfile: in.lj.miniMD
    # Datafile: None
Physics Settings:
    # ForceStyle: LJ
    # Force Parameters: 1.00 1.00
    # Units: LJ
    # Atoms: 864000
    # System size: 100.78 100.78 100.78
    # Density: 0.844200
    # Force cutoff: 2.500000
    # Timestep size: 0.005000
Technical Settings:
    # Neigh cutoff: 2.800000
    # Half neighborlists: 0
    # Neighbor bins: 50 50 50
    # Neighbor frequency: 20
    # Sorting frequency: 20
    # Thermo frequency: 100
    # Ghost Newton: 1
    # Use intrinsics: 0
    # Do safe exchange: 0
    # Size of float: 8

Starting dynamics ...
Timestep T U P Time
0 1.440000e+00 −6.773368e+00 −5.019671e+00 0.000 100
7.310629e−01 −5.712170e+00 1.204577e+00 3.650

Performance Summary:
```

Threads	nsteps	natoms	t-total	t-force	t-neigh	t-comm	t-other
16	100	864000	3.64	2.58	0.73	0.14	0.18

1.3 The Chipyard framework

Chipyard is a framework for designing and evaluating full-system hardware using agile teams. It is composed of a collection of tools and libraries designed to provide an integration between open-source and commercial tools for the development of systems-on-chip. This framework makes it easier to design, integrate and simulate a custom **SoC** (i.e. System on a Chip): in other words, it aims to be the "one-stop shop" for creating and testing SoCs.

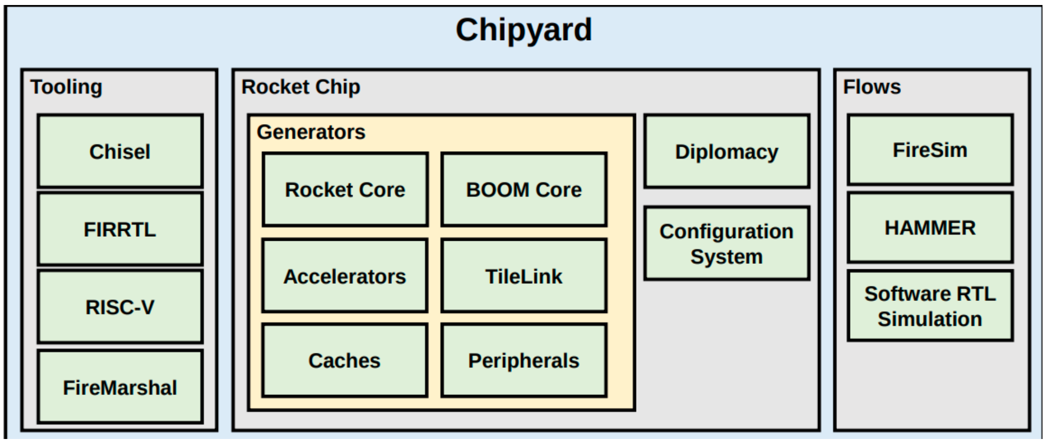

Figure 1.3. Large library of open-source projects for RISC-V SoC development included in Chipyard.

The trend towards agile hardware design and evaluation provides an ecosystem of debugging and implementation tools, that make it easier for computer architecture researchers to develop novel concepts. Chipyard hopes to build on this prior work in order to create a singular location to which multiple projects within the Berkeley Architecture Research can coexist and be used together.

1.3.1 Components

Let's briefly analyze the components that will be useful during the further discussions about this project's realization.

Generators

A Generator can be thought of as a generalized Register Transfer Level (RTL) design, written using a mix of meta-programming and standard RTL, which is, essentially, a single instance of a design coming from a generator (this type of meta-programming is enabled by the Chisel hardware description language). However, by using meta-programming and parameter systems, generators can allow for integration of complex hardware designs in automated ways. Chipyard builds the generators from source code each time (although the build system will cache results if they have not changed), so changes to the generators themselves will automatically be used when building with Chipyard and propagate to software simulation, FPGA-accelerated simulation, and VLSI flows.
It is crucial to notice that, whithin this framework, all of the Chisel RTL is written as

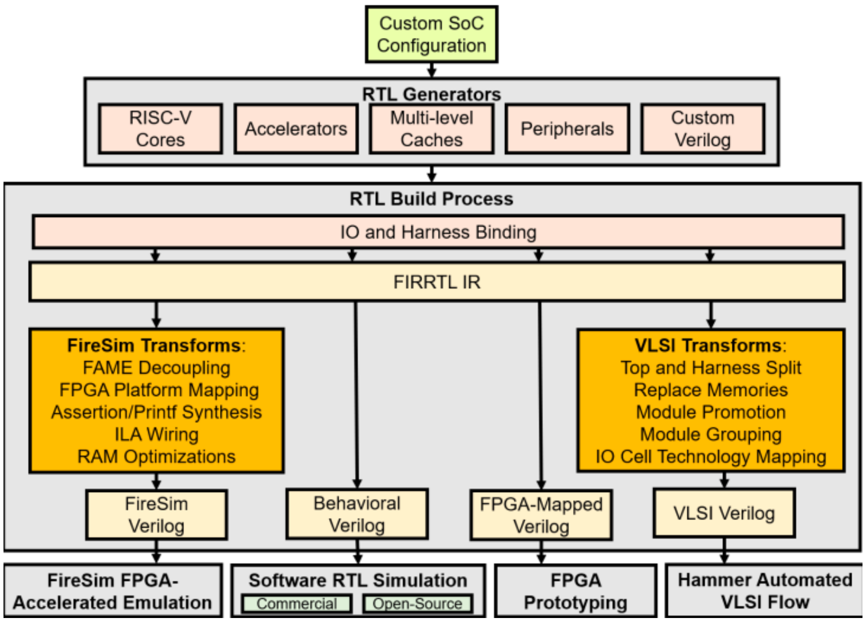
Custom SoC Configuration
RTL Generators
RISC-V Cores
Accelerators
Multi-level Caches
Peripherals
Custom Verilog
RTL Build Process
IO and Harness Binding
FIRRTL IR
FireSim Transforms: FAME Decoupling FPGA Platform Mapping Assertion/Printf Synthesis ILA Wiring RAM Optimizations
VLSI Transforms: Top and Harness Split Replace Memories Module Promotion Module Grouping IO Cell Technology Mapping
FireSim Verilog
Behavioral Verilog
FPGA-Mapped Verilog
VLSI Verilog
FireSim FPGA-Accelerated Emulation
Software RTL Simulation
Commercial
Open-Source
FPGA Prototyping
Hammer Automated VLSI Flow

Figure 1.4. Custom SoC configurations are integrated through generators.

generators: these parametrized programs, designed to generate RTL code based on configuration specifications, can be used to generate Systems-on-Chip (**SoCs**) using a collection of system components organized in unique generator projects.

The Chipyard Framework currently consists of the following RTL generators, as outlined in the following table:

Processor cores	Accelerators	System components
Rocket Core	Hwacha	icenet
BOOM	Gemmini	sifive-blocks
CVA6 Core	SHA3	AWL
Ibex Core		testchipip

Tools

In order to create new RTL designs quickly, the framework supports several tools: one of the most common is the Chisel Hardware Construction Language and the FIRRTL Compiler. Chipyard, indeed, is written in Chisel, an embedded language within Scala that provides a set of libraries to help hardware designers create highly parameterizable RTL. It is used to write RTL generators using meta-programming, by embedding hardware generation primitives in the Scala programming language. The Chisel compiler elaborates the generator into a FIRRTL output.

On the other hand, FIRRTL is an intermediate representation of the circuit we are going to generate. As previously stated, it is emitted by the Chisel compiler and, through the FIRRTL compiler, Chisel source files are translated into another representation, such as Verilog. This intermediate step will allow the user to run FIRRTL passes that can primarily do dead code elimination, circuit analysis and connectivity checks. These two tools in combination allow quick design space exploration and development of new RTL.

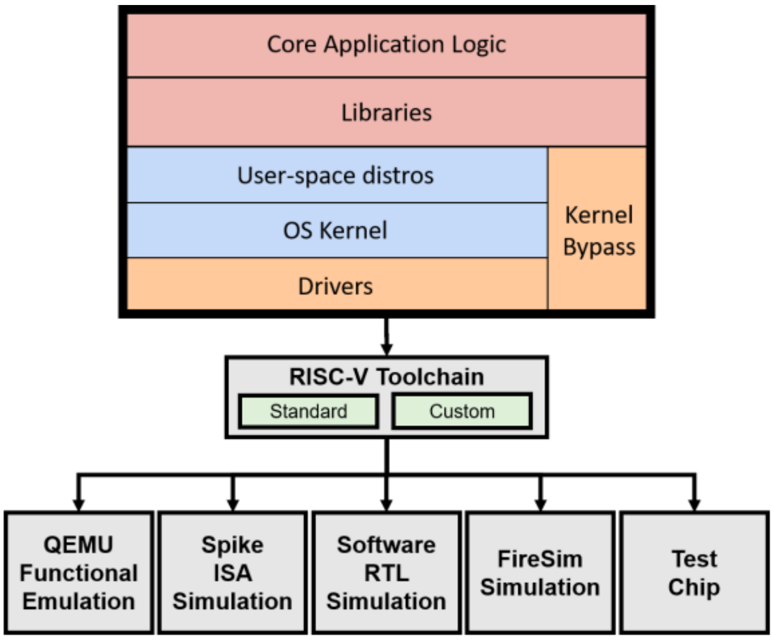

Figure 1.5. Custom Software handling

Toolchains

Chipyard involves two main toolchains, **riscv-tools** and **esp-tools**. The former is a collection of software toolchains used to develop and execute software on the RISC-V ISA. It includes the compiler and assembler toolchains, functional ISA simulator (spike), the Berkeley Boot Loader (BBL) and proxy kernel.
The latter is a fork of riscv-tools, designed to work with the Hwacha non-standard RISC-V extension. This fork can also be used as an example demonstrating how to add additional RoCC accelerators to the ISA-level simulation (Spike) and the higher-level software toolchain (GNU binutils, riscv-opcodes, etc.).

Simulators

Chipyard supports two classes of simulation:

- Software RTL simulation using commercial (VCS) or open-source (Verilator) RTL simulators
- FPGA-accelerated full-system simulation using FireSim

To the first category belong Verilator and VCS. Verilator is an open source Verilog simulator. The verilator directory provides wrappers which construct Verilator-based simulators from relevant generated RTL, allowing for execution of test RISC-V programs on the simulator (including vcd waveform files). VCS, instead, is a proprietary Verilog simulator. It provides wrappers which construct VCS-based simulators from relevant generated RTL, allowing for execution of test RISC-V programs on the simulator.
On the other hand, an example on a FPGA-accelerated full-system is FireSim, an open-source cycle-accurate hardware simulation platform that runs on cloud FPGAs

(Amazon EC2 F1). FireSim allows RTL-level simulation at orders-of-magnitude faster speeds than software RTL simulators. FireSim also provides additional device models to allow full-system simulation, including memory models and network models.
Dealing with performances, software RTL simulators of Chipyard designs run in the order of1 KHz, but compile quickly and provide full waveforms; conversely, FPGA-accelerated simulators run in the order of 100 MHz, making them appropriate for booting an operating system and running a complete workload, but have multi-hour compile times and poorer debug visibility.

Prototyping

FPGA prototyping is supported in Chipyard using SiFive's fpga-shells. Some examples of FPGAs supported are the Xilinx Arty 35T and VCU118 boards.

Chapter 2

RISC Core Configurations

In this chapter we will introduce and analyze the *generators* we have been using during the whole testing and benchmarking process. First of all, we will discuss about Rocket Chip, an open-source SoC infrastructure: then, we will introduce Rocket Core and BOOM, custom cores through which we carried out the simulations.

2.1 Rocket Chip

Rocket Chip generator is a SoC generator developed at Berkeley. Chipyard uses the Rocket Chip generator as the basis for producing a RISC-V system on a chip.
Note that Rocket Chip is distinct from Rocket core, which is just one building block of the entire SoC infrastructure. Rocket Chip includes many parts of the SoC besides the CPU: though it uses Rocket core CPUs by default, it can also be configured to use the BOOM out-of-order core generator or some other custom CPU generator instead.

The diagram 2.1 shows a dual-core Rocket system. Each Rocket core is grouped with a page-table walker, L1 instruction cache, and L1 data cache into a RocketTile. The Rocket core can also be swapped for a BOOM core 2.3. Each tile can also be configured with a RoCC accelerator that connects to the core as a coprocessor.
The tiles connect to the SystemBus, which connect it to the L2 cache banks. The L2 cache banks then connect to the MemoryBus, which connects to the DRAM controller through a TileLink to AXI converter.
The Debug Unit is used to control the chip externally: furthermore, it can be used to load data and instructions to memory or pull data from memory. It can be controlled through a custom DMI or standard JTAG protocol.

2.2 Rocket Core

Rocket Core is a 5-stage in-order scalar processor core generator. The Rocket core is used as a component within the Rocket Chip SoC generator: it is a core combined with L1 caches (data and instruction caches) form a Rocket tile, which is the replicable component of the Rocket Chip SoC generator. The Rocket core supports the open-source RISC-V instruction set and is written in the Chisel hardware construction language. It has an memory management unit (MMU) that supports page-based virtual memory, a non-blocking data cache, and a front-end with branch prediction.

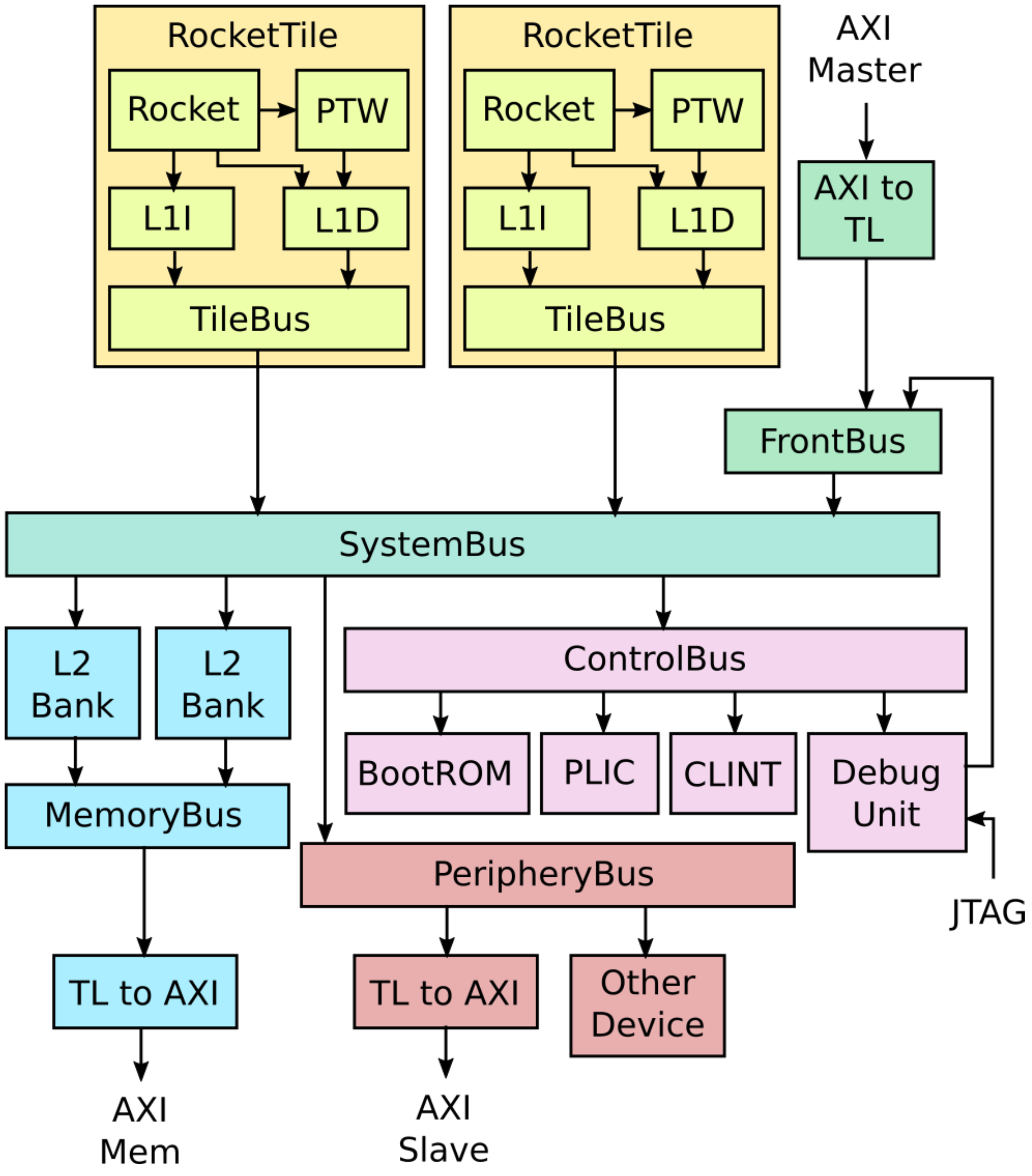

Figure 2.1. Diagram of a typical Rocket Chip system.

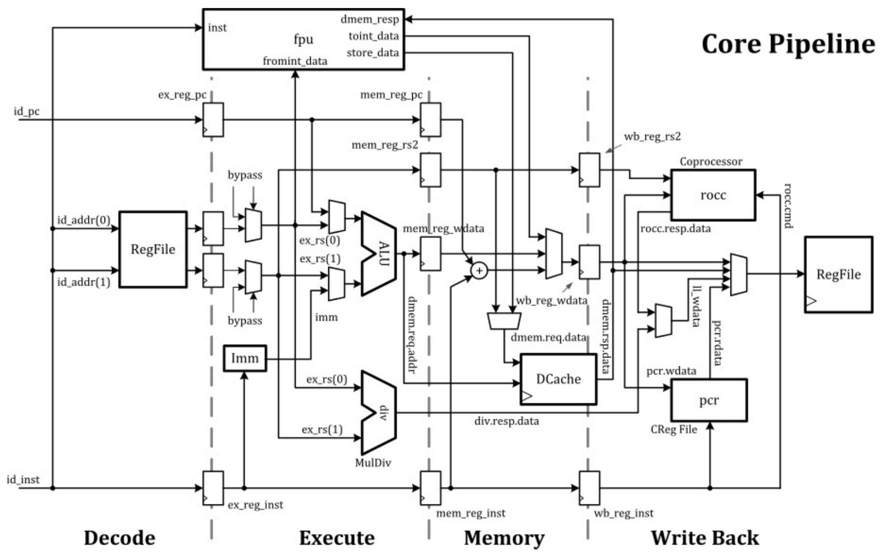

Figure 2.2. Rocket core pipeline.

Branch prediction is configurable and provided by a branch target buffer (BTB), branch history table (BHT), and a return address stack (RAS). For floating-point, Rocket makes use of Berkeley's Chisel implementations of floating-point units. Rocket also supports the RISC-V machine, supervisor, and user privilege levels. A number of parameters are exposed, including the optional support of some ISA extensions (M, A, F, D), the number of floating-point pipeline stages, and the cache and TLB sizes.

2.3 Berkeley out of order Machine

The Berkeley Out-of-Order Machine (**BOOM**) is a synthesizable and parameterizable open source RISC-V core written in the Chisel hardware construction language. It serves as a replacement to the Rocket core given by Rocket Chip (replaces the RocketTile with a BoomTile). BOOM is heavily inspired by the MIPS R10k and the Alpha 21264 out-of-order processors. Like the R10k and the 21264, BOOM is a unified physical register file design (also known as "explicit register renaming").
Conceptually, BOOM is broken up into 10 stages: Fetch, Decode, Register Rename, Dispatch, Issue, Register Read, Execute, Memory, Writeback and Commit. However, many of those stages are combined in the current implementation, yielding seven stages: Fetch, Decode/Rename, Rename/Dispatch, Issue/RegisterRead, Execute, Memory and Writeback (commit occurs asynchronously, so it is not counted as part of the "pipeline").

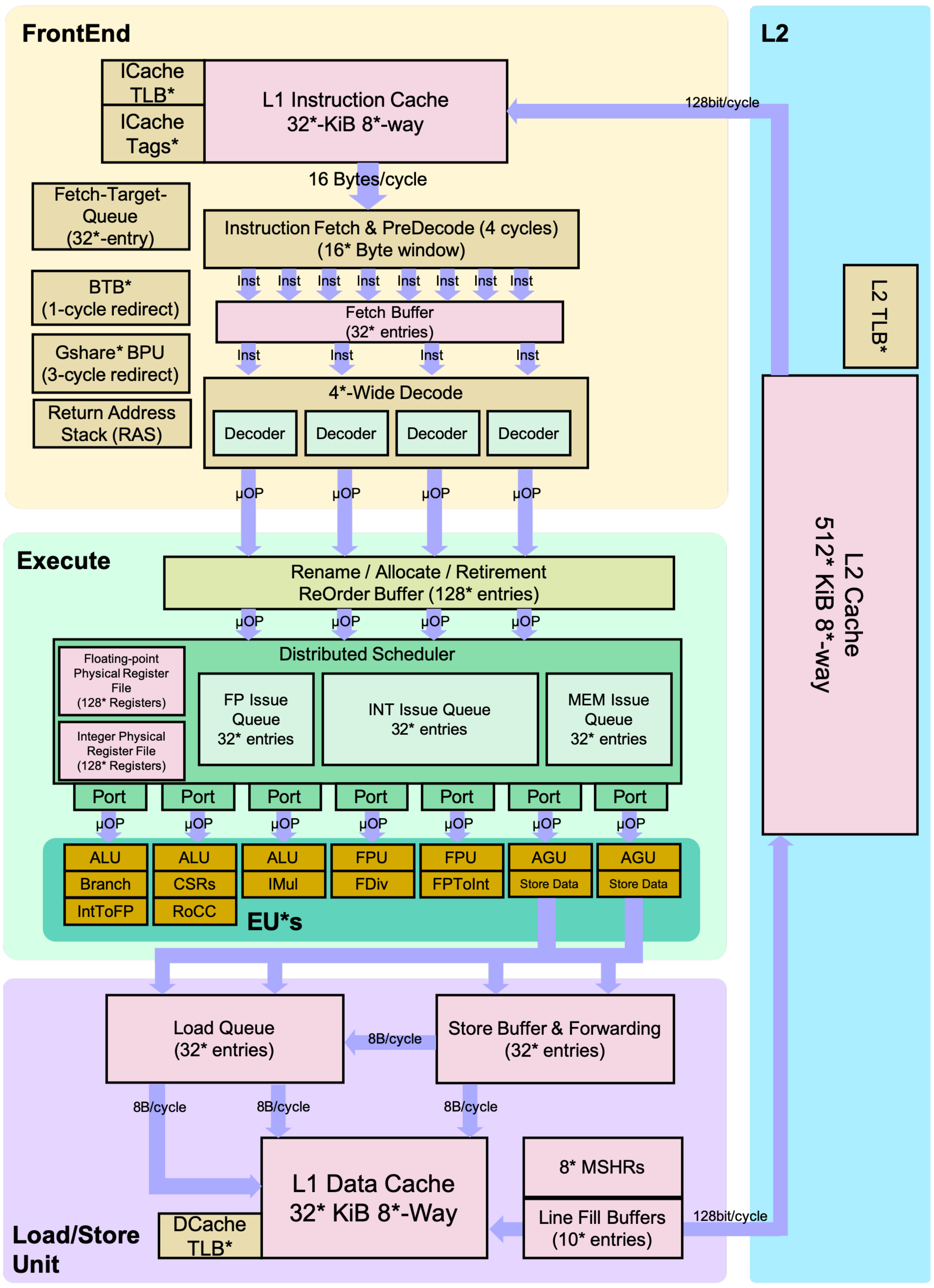
FrontEnd
ICache TLB*
ICache Tags*
L1 Instruction Cache 32*-KiB 8*-way
128bit/cycle
16 Bytes/cycle
Fetch-Target-Queue (32*-entry)
BTB* (1-cycle redirect)
Gshare* BPU (3-cycle redirect)
Return Address Stack (RAS)
Instruction Fetch & PreDecode (4 cycles) (16* Byte window)
Inst
Fetch Buffer (32* entries)
4*-Wide Decode
Decoder
µOP
Execute
Rename / Allocate / Retirement ReOrder Buffer (128* entries)
Distributed Scheduler
Floating-point Physical Register File (128* Registers)
Integer Physical Register File (128* Registers)
FP Issue Queue 32* entries
INT Issue Queue 32* entries
MEM Issue Queue 32* entries
Port
ALU
Branch
IntToFP
CSRs
RoCC
IMul
FPU
FDiv
FPToInt
AGU
Store Data
EU*s
Load Queue (32* entries)
8B/cycle
Store Buffer & Forwarding (32* entries)
L1 Data Cache 32* KiB 8*-Way
DCache TLB*
8* MSHRs
Line Fill Buffers (10* entries)
Load/Store Unit
L2
L2 TLB*
L2 Cache 512* KiB 8*-way

Figure 2.3. Detailed BOOM Pipeline.

Chapter 3

Methodology

In this chapter we will present the process we went through in order to extract the core function in charge of computing the delta of the forces among each atom and its neighbors. We will also investigate how we succeeded in simulating through Verilator, and which were the limitations that this tool showed during this analysis.

3.1 Function isolation

The first step we need to take was deciding which function could be the best candidate function to isolate and simulate. One of the most critical aspect of MD simulation is, clearly, the computation of the forces that link each atom with its neighbors: this aspect led us to choose and extract this particular fraction of the whole miniMD code. The class *force_lj*, moreover, contains different functions which actually perform this computation:

- **compute_original**: the original version of force compute in miniMD, which is not vectorizable;
- **compute_halfneigh**: optimized version of compute, which enables vectorization by getting rid of 2d pointers;
- **compute_fullneigh**: optimized version of compute, enables vectorization, uses pragma and use full neighbor lists

We choose to isolate the first version of this function (compute_original), in order to focus mainly on the basic computations carried out.

3.1.1 First step: set up a custom project

After choosing the function to extract, we needed to include the main headers:

- **atom.h**: includes the atom struct;
- **neighbor.h**: includes the neighbor struct;
- **types.h**: includes other useful types for the computation.

Then, we extracted the *compute_original* function, which, after setting up the parameters that will be involved in the computation, loops over all neighbors of each atom, storing forces on both atoms involved in each computation. The code below shows the main aspects of this function.

```
void compute_original(Atom atom, Neighbor neighbor, int me)
  //SETTING UP FORCES
  double cutforce = 2.5;
  int use_oldcompute = 0;
  ...

  //SETTING UP THE PARAMETERS OF THE FUNCTION
  int nlocal = atom.nlocal;
  int nall = atom.nlocal + atom.nghost;
  ...

  // CORE COMPUTATIONS
  for(int i = 0; i < nlocal; i++) {
    const int* const neighs = &neighbor.neighbors[i * neighbor.maxneighs];
    const int numneigh = neighbor.numneigh[i];
    const double xtmp = x[i * PAD + 0];
    const double ytmp = x[i * PAD + 1];
    const double ztmp = x[i * PAD + 2];
    const int type_i = type[i];
    for(int k = 0; k < numneigh; k++) {
      const int j = neighs[k];
      const double delx = xtmp - x[j * PAD + 0];
      const double dely = ytmp - x[j * PAD + 1];
      const double delz = ztmp - x[j * PAD + 2];
      int type_j = type[j];
      const double rsq = delx * delx + dely * dely + delz * delz;
      const int type_ij = type_i*ntypes+type_j;
      if(rsq < cutforcesq[type_ij]) {
        const double sr2 = 1.0 / rsq;
        const double sr6 = sr2 * sr2 * sr2 * sigma6[type_ij];
        const double force = 48.0 * sr6 * (sr6 - 0.5) * sr2 * epsilon[type_ij];
        f[i * PAD + 0] += delx * force;
        f[i * PAD + 1] += dely * force;
        f[i * PAD + 2] += delz * force;
        f[j * PAD + 0] -= delx * force;
        f[j * PAD + 1] -= dely * force;
        f[j * PAD + 2] -= delz * force;
        //compute the deltas
        eng_vdwl += (4.0 * sr6 * (sr6 - 1.0)) * epsilon[type_ij];
        virial += (delx * delx + dely * dely + delz * delz) * force;
      }
    }
```

3.1.2 Second step: input generation

Once set up the environment, we dealt with the problem of generating an input whose size could allow us to simulate it in our environment and with the tool we chose for this project. To achieve this goal, we modified the main function of miniMD, so that, at the end of the computation, the main elements of the simulation (atoms, neighbors, forces etc.) would be printed in a file, which would then used as a valid input for our custom environment. We also modified the input file of miniMD, in order to change the size of the problem (i.e. the number of atoms, the initial temperature etc.). The code below is an example of a valid input for the program (it is, actually, the configuration we chose to take into account).

```
lj              units (lj or metal)
none            data file (none or filename)
lj              force style (lj or eam)
1.0 1.0         LJ parameters (epsilon and sigma)
1 1 1           size of problem
10               timesteps
0.005           timestep size
1.44            initial temperature
0.8442          density
10              reneighboring every this many steps
2.5 0.30        force cutoff and neighbor skin
100             thermo calculation every this many steps
```

3.1.3 Limitations of the simulator: Verilator

It is important to notice that the simulator we choose to test our function, Verilator, showed some limitations during the project. The main issues we went through were the following:

- **input file**: when trying to manage files (e.g. opening a file and importing its content), we were not able to carry out the simulation. For this reason, in order to bypass this problem, we decided to exploit two python scripts, **read_atom.py** and **read_neigh.py**. These scripts read the file in input and produce two headers, containing all the variables needed for the computations. As this operations are performed before the simulation takes place, we achieved our result without changing the input of the program.

- **array dimensions**: when trying to import array over a certain dimension, the simulation broke. This is another limitation of the tool, which forced us to resize the problem multiple times, in order to find the right balance of atoms, neighbors and force among them

This is due to the fact that the simulator, actually, will not be able to exploit real syscalls, since no kernel is provided under this architecture. Nevertheless, this feature will not invalidate in any way the results we achieved, as showed in the previous points. After everything was set up, we built a *makefile*, to nimbly compile the main function.

Chapter 4

Experiment configurations

In this chapter we will delve into details of our analysis: in the first section, we will discuss about the setting we chose for the system to work in the proper way (in order to carry out a sensible simulation). Then, we will show the actual configurations of the system that we tested and developed, explaining the line of reasoning which led us to provide such changes.

4.1 System parameters and data structures

In the table below are summarized the main settings of the system, which are taken as input for our system (and are produced as output through a simulation of *miniMD*).

General settings	
Number of atoms	200
Number of local atoms	200
Number of ghosts	80
Types of atoms	4
Max number of atoms	364
Max number of neigh	20
Epsilon	1.0

Continued on next page

Table 4.1 – *Continued from previous page*

Sigma	1.0
Cutforce	2.5
Virial	1.4822e-323
Mass	1

The first parameters (number of atoms, local, ghost and type) are the most crucial ones, which determine the size of the problem. The other parameters (epsilon, sigma etc.) refer to different characteristic of the force that will be computed at each iteration of the main loop.
Other input data, as the position, the force and the velocity of each atom, are read as inputs and stored in proper data structures, as synthesized in the following table.

Table 4.2. Main data structures used for the simulation

Data structure	**Description**
long unsigned int* x	the position of each atom
long unsigned int* v	the velocity of each atom
long unsigned int* f	the force associated to each atom
int* type	the type of each atom
int* f_numneigh	the number of neighbors of each atom
int* f_neighbors	the actual neighbors of each atom

All these parameters can either be read from an input file (produced by running a simulation of *miniMD*) or, through the support of the python scripts *read_atoms.py*

and *read_neighs.py*, can be imported as header files.
It is important to remind that this simulation, for testing purposes, represent a toy example, since a standard miniMD simulation is run with approximately 900.000 atoms.

4.2 Architecture configurations

4.2.1 Experimental Setup

Let's now analyze the different architectures we took into account. It is crucial to remark that, during the whole simulation phase, we ensured that the result of the various computations were correct and coherent with the system we were testing. Therefore, the result of the computations were the following, where **eng_vdwl** and **virial** are the results of the computation.

```
 # Computation settings:
        > Natoms: 200
        > Nlocal: 200
        > Nghost: 80
        > x[0]: 1.61
        > v[0]: -1.4
        > f[0]: 19.31
        > type[0]: 3
 # Done ...
        > Nmax: 364
        > Maxneighs: 20
 # Done ...
 # Setting up force :
        > epsilon: 1.0
        > sigma: 1.0
        > cutforce: 2.50

**First cc of main loop : 110102 //depends on the actual configuration
**Last cc of main loop : 146396  //depends on the actual configuration
 # End of computation
        >Eng_vdwl: -15.17
        >Virial: 485.80
```

Furthermore, we decided to focus on the Rocket Chip architecture, modifying its parameters and, above all, the core that make it up. In particular, we benchmark and tested the performance of the whole architecture with Rocket Core, BOOM and, finally, various custom BOOM variants.

4.2.2 Rocket Core

The first simulations were carried out trough the Rocket Core configuration (2.2). Below it is shown the main features (and parameters) of the chosen configuration.

```
class WithNBigCores(n: Int = 1) extends Config((site, here, up) => {
  case RocketTilesKey => {
    val prev = up(RocketTilesKey, site)
    val idOffset = overrideIdOffset.getOrElse(prev.size)
    val big = RocketTileParams(
      core   = RocketCoreParams(mulDiv = Some(MulDivParams(
        mulUnroll = 8,
        mulEarlyOut = true,
        divEarlyOut = true))),
      dcache = Some(DCacheParams(
        rowBits = site(SystemBusKey).beatBits,
        nMSHRs = 0,
        blockBytes = site(CacheBlockBytes))),
      icache = Some(ICacheParams(
        rowBits = site(SystemBusKey).beatBits,
        blockBytes = site(CacheBlockBytes))))
    List.tabulate(n)(i => big.copy(hartId = i + idOffset)) ++ prev
  }
})
```

As stated in section (2.2), this is an example of an **in-order execution** which basically means that instructions are fetched, executed and completed in compiler-generated order, so statically scheduled. if one instruction need to stall, since it is waiting for a parameter which is not already there, all the next instructions will be stalled as well. One of the greatest difficulty of in-order execution is in the conditional and jump instructions: since this will be executed when the condition occurs, the entire code execution will be heavily slowed down.

4.2.3 Boom

We then decided to swap the Rocket Core with another type of generator, i.e. the Berkeley Out-Of-Order Machine (BOOM, 2.3). We tested and benchmarked all the main configuration:

- **WithNSmallBooms**
- **WithNMediumBooms**
- **WithNLargeBooms**
- **WithNMegaBooms**
- **WithNGigaBooms**

In this case, we were testing an out-of-order execution: the way in which the core execute the code is thought to avoid the execution stops. Indeed, instructions are executed in a different order than those indicated in the code: this is made possible since each type of instruction has a type of execution unit assigned to it and, depending on the type of instruction, the core exploits the specific type of execution unit needed.

More precisely, instructions are fetched in compiler-generated order, while their completion may be in-order or out-of-order: in between they may be executed in some other order, since independent instructions behind a stalled instruction can pass it (**dynamic execution**).

4.2.4 CustomBoom

We then decided to modify the BOOM architecture, in order to boost the performance of our computations. We focused on modifying the main parameters of the architecture, which could significantly have an impact on the clock cycles needed to execute the main loop of our function. The parameters that we have been modifying, along with their description, are summarized in the table below.

Table 4.3. Architecture's main parameters

Parameter	**Description**
fetchWidth	number of instructions that will be fetched at once
decodeWidth	number of instructions that will be decoded at once
numRobEntries	number of entries of the ROB buffer
issueParams	issue queue types and units
numIntPhysRegisters	number of integer physical registers
numFpPhysRegisters	number of floating point physical registers
numLdqEntries	number of entries in the load queue
numStqEntries	number of entries in the store queue
enablePrefetching	bool to allow prefetching
enableBranchPrediction	bool to allow branch prediction

The final configuration we achieved is shown (in its main aspects) in the following code snippet:

```
class WithNCustomBooms(n: Int = 1,) extends Config(
   ...
   BoomTileAttachParams(
    tileParams = BoomTileParams(
        core = BoomCoreParams(
          fetchWidth = 8,
          decodeWidth = 4,
          numRobEntries = 128,
          issueParams = Seq(
                IssueParams(issueWidth=2, numEntries=24, dispatchWidth=4),
                IssueParams(issueWidth=2, numEntries=20, dispatchWidth=4),
                IssueParams(issueWidth=6, numEntries=58, dispatchWidth=4)),
          numIntPhysRegisters = 128,
          numFpPhysRegisters = 128,
          numLdqEntries = 32,
          numStqEntries = 32,
          maxBrCount = 20,
          numFetchBufferEntries = 32,
          enablePrefetching = true,
          ftq = FtqParameters(nEntries=40),
          fpu = Some(rocketchip.FPUParams(sfmaLatency=1, dfmaLatency=1)),
        dcache = Some(
          DCacheParams( nSets=64, nWays=32, nMSHRs=16, nTLBWays=32)),
        icache = Some(
          ICacheParams( nSets=64, nWays=32, fetchBytes=4*4)),)
```

4.3 Other settings

Considering the architecture choices we showed above, we tried to run the simulation first with an insignificant amount of atoms (i.e. 4), then with a still toy example, but better calibrated system (the one analyzed in section 4.2).
With regard to the compiling phase, we first compiled the binary without any type of optimization, and then added the option **-O3**. Changing the value of the variable "-O", allows to modify (and improve, in some cases) the overall level of code optimization. Changing this value will result, often, in longer compile time and larger memory usage, especially as the optimization level is raised. "-O3" is the highest optimization level: costly optimizations are activated from the point of view of compile time and memory usage, while it is not guaranteed that such operations will reflect to an improvement in the code performance. It also enables *-ftree-vectorize* so that loops in the code become vectorized and AVX YMM registers will be used.

Chapter 5

Experiment results

In this chapter we will show the results we obtained through the different experiment configurations we showed in section 5. The results are obtained enabling the optimization -*O3*, and considering the system configurations analyzed in section 4.2.

5.1 Entire Program evaluation

First of all, let's analyze how the execution of the whole program changed along with the different architectures, by taking into account the clock cycles needed to perform the entire computation.

Table 5.1. Clock cycles needed for the computation of the whole program over different architectures

Architecture	**Clock cycles (whole program)**
RocketConfig	296'685
SmallBoomConfig	264'561
MediumBoomConfig	245'012
LargeBoomConfig	221'576
MegaBoomConfig	212'300
CustomBoomConfig	210'414

As we could observe, even if we entirely changed the core and the main parameters through which the simulation was carried out, the performance did not drastically change: this was due to the fact that our code included, besides the core functionality (force computation), many other computations, needed to set up the inputs and the whole system (variables, initializations etc.). This analysis led us to perform a more accurate benchmark, only considering the main steps needed to perform the computation of the forces.

5.2 Main computations evaluation

The following table summarizes the performance changes, with regard to the specific loop involved in the force computations.

Table 5.2. Clock cycles needed for the computation of the main loop over different architectures

Architecture	**Clock cycles (main loop)**
RocketConfig	36’294
SmallBoomConfig	26’788
MediumBoomConfig	22’330
LargeBoomConfig	18’378
MegaBoomConfig	13’581
CustomBoomConfig	9’256

At this point we were able to appreciate the performance changes, due to the different settings and parametrizations. As we can observe, the main loop needed 10.000 less than RocketConfig, when simulated through the first (and basic) configuration of BOOM. This value decreases drastically simulating the system with our *CustomBoom-Configuration*, which, optimizing the main parts of the code, will need 9.256 clock cycles to perform the core computations.

5.2.1 RocketCore

Simulating the system with Rocket Core, we were able to obtain crucial information about the instructions executed (their number and frequency) and the clock cycles needed to execute them. It is important to underline that the data that we will provide in the next pages refers to the main functions of the system (those regarding the computation of the forces): this sharper analysis enabled us to focus on the most critical part of the computation, the one that, with a growing amount of atoms, would lead our simulation's performance to get worse.
The pie chart shown below should provide a useful overview: as we can observe, the number of integer and floating point operations is roughly the same, while the clock cycles needed to execute those two type of instructions radically changes.

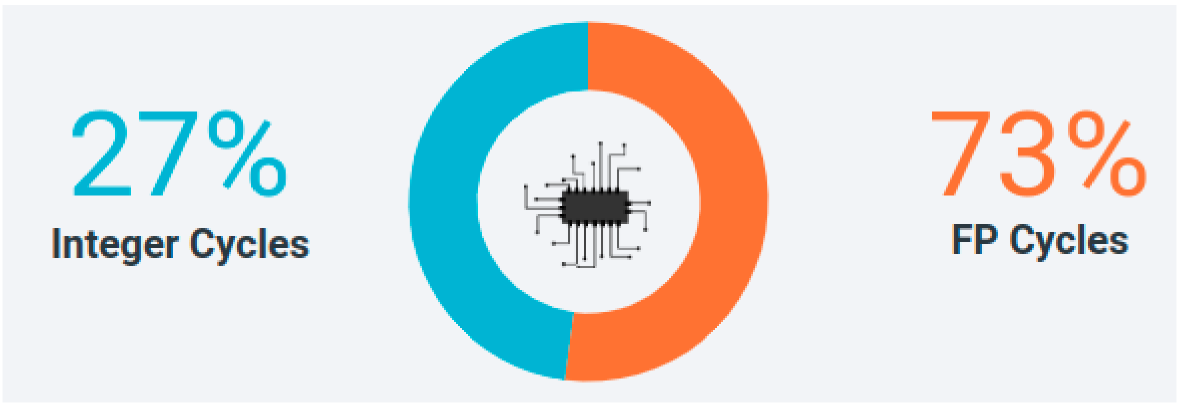

Figure 5.1. Comparison between the total amount of clock cycles needed to execute integer and floating point instructions.

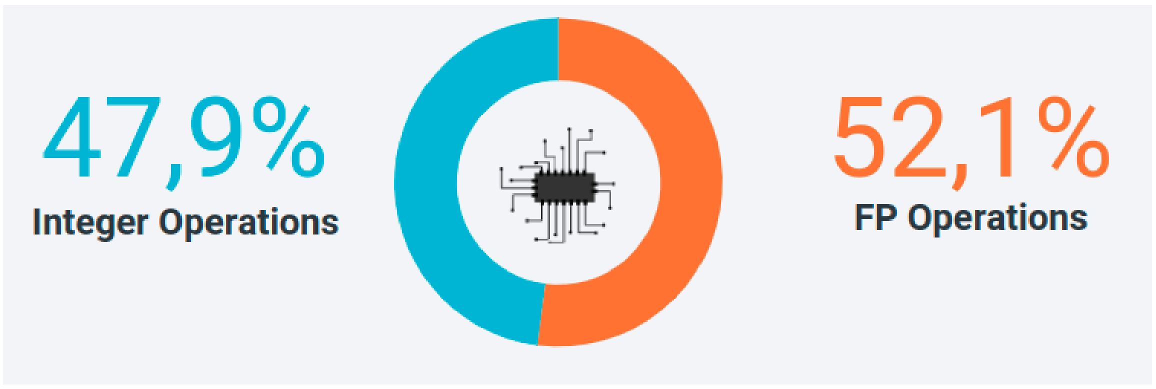

Figure 5.2. Comparison between the number of integer and floating point instructions executed.

We also focused on the instruction mix (i.e. the blend of instruction types in the disassembly of the binary code): as highlighted by the chart, the **integer add** instruction is relevant, as well as the floating point multiplication and add.
Another important statistic we analyzed is the total amount of clock cycles needed to perform each type of operation.

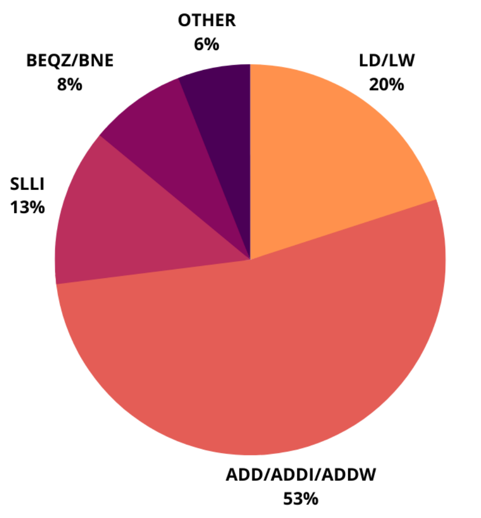

Figure 5.3. Percentage of integer instruction occurrence, over the total amount of integer instructions (based on the instruction mix of the RocketCore architecture)

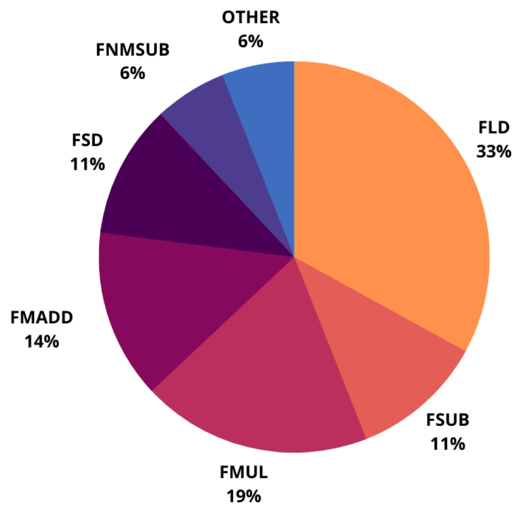

Figure 5.4. Percentage of floating point instruction occurrence, over the total amount of integer instructions (based on the instruction mix of the RocketCore architecture)

The charts 5.5 and 5.6 show that, even if the number of instructions was almost equally distributed between integer and floating point instruction, the same cannot be asserted for the distribution of clock cycles. In this specific case, the floating point operations are much more relevant (70% of the total amount of clock cycles), and some types of instructions (such as the floating point division or multiplication) take on a much more central role, in the whole analysis.

5.2.2 BOOM

After analysing the performance of each BOOM core implementation, we tried ourselves to change the core parameters in order to improve our system's computations. The

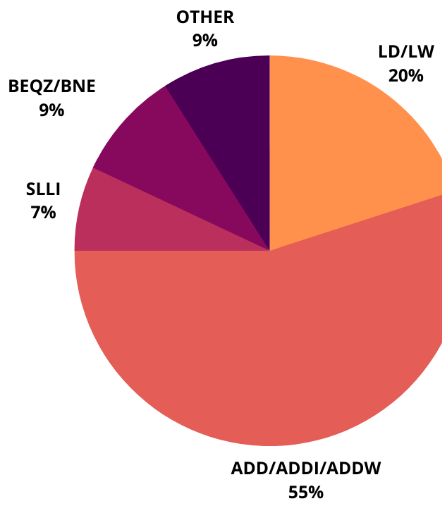

Figure 5.5. Percentage of clock cycles per integer instruction, over the total amount of clock cycles for integer instructions

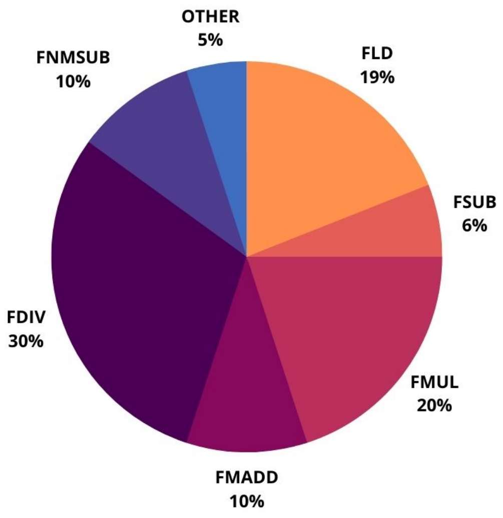

Figure 5.6. Percentage of clock cycles per FP instruction, over the total amount of clock cycles for FP instructions

next table summarizes the workflow we followed, in order to reach the model described in section 4.2.4. Note that each modification or percentage in considered with regard to the **MegaBoom** configuration.

Parameter(s)	**Old value - New value**	**Performance change (%)**
IssueWidth(INT FU)	4 → 6	↓1%
IssueWidth(INT FU)	4 → 8	↓1%
IssueWidth(INT FU)	4 → 10	↓1%
IssueWidth(FP FU)	2 → 4	↑4%
IssueWidth(FP FU)	2 → 6	↑16%
IssueWidth(FP FU)	2 → 8	↑16%
IssueWidth(FP FU)	2 → 10	↑16%

As we could expect from the instruction mix, a crucial role is played by the FP instructions: when boosting the number of the FP functional units involved in the computations, the performance of our system get better, while increasing the number of integer functional units will not give the computation any advantage. Let's now focus on the other parameters shown in section 4.2.4 and how their changes have reflected on a change on the performance of our architecture. The next table is built in an incremental way: the percentage of performance change is considered with reference to the best configuration showed in the previous table. It is necessary to highlight that we got an improvement in performance while decreasing the **sfma** and **dfma** latency: this datum should be analyzed carefully, since it is not always the case that, decreasing the latency, an improvement involves the whole computation. Indeed, many factors are influenced by such a change, so, before applying this modification to a further version of this architecture, it should always be checked the effective behaviour of the computation.

Parameter(s)	**Old value - New value**	**Performance change(%)**
Branch prediction	True → False	↓59.5%
Ldq, Stq Entries	32 → 40	↓2%
fetchBufEntries	32 → 40	↓3%
sfma, dfma latency	4 → 8	↓22%
sfma, dfma latency	4 → 2	↑7.7%

Continued on next page

Table 5.4 – *Continued from previous page*

sfma, dfma latency	$4 \rightarrow 1$	↑13.5%
DCache, ICache (nWays)	$8 \rightarrow 16$	↑4.2%
DCache, ICache (nWays)	$8 \rightarrow 32$	↑5%
DCache, ICache (nWays)	$8 \rightarrow 64$	↑4.4%
DCache, ICache (nMSHRs)	$8 \rightarrow 16$	↑5.5%
DCache, ICache (nMSHRs)	$8 \rightarrow 32$	↑4%

All these available data were exploited in order to develop the Custom architecture analyzed in section 4.2.4.

Chapter 6

Future developments

This project succeeded in its goal of improving the performance (in terms of clock cycles) of a specific computation, and benchmarking the different results we obtained through the simulation of different architectures, with different structures and parameters. In order to go beyond these improvements, it could be developed a specialized FU, able to perform such computations in a custom, specialized and optimized way. An interesting idea could be to drop the "neighbor list" approach, which already showed its limitations, and led to the development of newer versions of the function *compute_original* we analyzed during this project. Essentially, at each iteration, we would consider the current atom and the **filtered list** of *all* the other atoms: this filtering is based on the distance between the current atom and the one being considered among all the other ones. If the distance between them is under a certain threshold, the couple will be take into account for the force computation, otherwise it won't be even calculated. This filter should be a specialized HW component, which could boost the whole computation.