POLITECNICO DI TORINO

Collegio di Ingegneria Informatica,
del Cinema e Meccatronica

Corso di Laurea Magistrale in Software Engineering

Tesi di Laurea Magistrale

Analysis and dynamic optimization of energy consumption on HPC applications based on real-time metrics

Relatore:
prof. Matteo Sonza Reorda

Candidato:
Fabio Ferrero

Ottobre 2017

Abstract

Over the past 30 years, both hardware and operating systems have become incredibly complex and efficient, both in term of computing performances and energy consumption. However, in the High Performance Computing (HPC) field new hard challenges are coming regarding systems energy consumption.

In last decades the demand for high performances has continuously increased, boosting the growth of HPC systems, up to enormous dimensions: as an example, the first two machines from the TOP500 list [1], called Sunway and Tianhe-2, have respectively 10 million and 3 million cores, with a power consumption of 15 and almost 18 mega-watts. Sunway, the first one, is close to 100 Peta-Flops of computing power.

Nowadays we are facing the challenge of Exascale computing: in a few years, humanity will be able to build the first supercomputer with 1 Exa-Flops of computing power, ten times more powerful than Sunway. Such a computing power will be incredibly useful in many domains, like for mathematical, industry and artificial intelligence simulations.

But how much energy will consume such a powerful machine? Ten times more than Sunway would be around 150 mega-watts. This power consumption is not sustainable, in terms of energy cost, power supply facility and environmental impact. We need then to design hardware and software that consume less being more powerful.

In this thesis, we present Bull Dynamic Power Optimizer (BDPO), a software solution that aims to reduce the energy consumption of a parallel application running on an HPC system. Our approach is to optimize the application at runtime. BDPO uses Dynamic Voltage Frequency Scaling and phase detection by monitoring real-time metrics retrieved through hardware performance counters.

We will describe our hardware counters study on micro-benchmarks, the BDPO design, and test on real HPC applications, where we reach up to 11% of energy gain with less than 1% performance loss.

Acknowledgements

At the end of my internship experience, I really would like to thank my managers Andry Razafinjatovo and Valérie Favier for giving me, in cooperation with the Grenoble INP and Politecnico di Torino schools, this amazing opportunity. In particular, I would like to thank Andry for his incredible enthusiasm, vision and inspiration.
I would like to thank Abdelhafid Mazouz for all the patience, the time and the experience he put to help us in our work.
I would like to thank Jean-François Méhaut and Matteo Sonza Reorda for their supervision and very helpful tips.
It was a pleasure to work with the Power Efficiency team at Bull: Franck, Marc, Hafid, Bertrand, Van Dung and Yoanne. Thanks for all your support and the work done together.
Last but absolutely not least, I would like to thank Mathieu Stoffel, my internship mate, for his incredible patience for my french speaking and his friendship, besides his technical skills and amazing ping-pong time spent together.

Contents

List of Abbreviations

ACPI	**A**dvanced **C**onfiguration and **P**ower **I**nterface
API	**A**pplication **P**rogramming **I**nterface
BDPO	**B**ull **D**ynamic **P**ower **O**ptimizer
BIOS	**B**asic **I**nput **O**utup **S**ystem
DVFS	**D**ynamic **V**oltage **F**requency **S**caling
GEOPM	**G**lobal **E**xtensible **O**pen **P**ower **M**anager
HDD	**H**ard **D**isk **D**rive
HPC	**H**igh **P**erformance **C**omputing
IPC	**I**nstructions **P**er **C**ycle
MAPE	**M**onitor-**A**nalyze-**P**lan-**E**xecute
MPI	**M**essage **P**assing **I**nterface
NFS	**N**etwork **F**ile **S**ystem
OS	**O**perating **S**ystem
OSPM	**O**perating **S**ystem **P**ower **M**anagement
PMU	**P**erformance **M**onitoring **U**nit
RAM	**R**andom **A**ccess **M**emory
RDTSC	**R**ea**D** **T**ime **S**tamp **C**ounter
REST	**R**untime **E**nergy **S**aving **T**echnology
SSD	**S**olid **S**tate **D**rive
TDP	**T**hermal **D**esign **P**ower

1 Introduction

Usually solving scientific problems leads to hard or long calculations. This happens very often in some particular domains, especially those that have to deal with numerical simulations. Computing systems that aim to efficiently solve those type of problems are called in general High Performance Computing (HPC) systems. Those clusters usually composed of a large number of nodes rely on powerful hardware and software facilities in order to use as effectively as possible program parallelism.

In last decades, HPC clusters have become increasingly powerful, reaching the computing power of around 100 Paeta-Flops. A real example is Sunway TaihuLight, the most powerful supercomputer according to the TOP500 list of June 2017 [1]. Obviously, to achieve such a computational power a great energy supply is required. Nowadays HPC supercomputers with this capability, like Sunway, consume around 15 mega-watts (for comparison, the charger of my personal computer can provide a power of 85W). This, by itself, is already a great amount of power consumption, but the power, both in terms of computing and consumption, of those systems is contiguously growing: the market is ready to come to Exascale computing in a few years. 1 Exa-Flops corresponds to 1000 peta-Flops, that roughly means that we need to build a supercomputer with ten times more computing power than modern machines. Such a supercomputer, by estimation, will consume ten times more energy: surely, it is not possible for cost and energy supply issues to have a supercomputer that consumes 150 mega-watts. In order to build a sustainable system in terms of energy cost and environmental impact, we would expect to budget at most around 20 to 30 mega-watts of power consumption, as set by the US Department of Energy as power constraint for Exascale machines [2]. This example clarifies the need of constant research on energy consumption of HPC systems, both for hardware and software effectiveness.

In this context, HPC clusters are used by a lot of research laboratories and companies in very different fields, such as physics, biology, meteo services, automotive engineering and oceanography, all domain that need some simulation or a lot of computations. Usually, in order to solve those type of problems, an expert on a specific domain define a theoretical model for the simulation. Then, knows that those models are highly parallelizable on a computing machine[1], either the expert himself or another programmer translates the scientific model into a parallel program, usually using low-level programming languages (C, C++, Fortran) and well-known libraries for parallel programming (MPI, OpenMP). The result of this process is what we call a

[1] In parallel theory, those problems usually reduce to big matrices operations, that are known from the literature to be, in most of the cases, highly parallelizable.

parallel application, that is usually part of the big domain of HPC applications. In this thesis, we face the problem of energy consumption optimization of those type of applications.

Our approach is then to reduce the energy consumption of real parallel applications without make any change in their source code. Moreover, our objective is to reach energy reduction while avoiding any performance degradation. We produced a system that computes real-time metrics to understand the compute node behavior. Relying on those metrics and phase detection, it changes CPU operating frequency dynamically in order to reduce power consumption only when needed. Testing our tool on real HPC applications, we reached up to 11% of energy savings with less than 1% performance loss.

This thesis was written during our six-month internship in the Energy Efficiency team of Bull s.a.s., an Atos company. All the presented material and results are the outcome of the work done by Mathieu Stoffel and Fabio Ferrero (the writer) during the internship, in collaboration with the other team members. Both of us produced his-own version of this manuscript, avoiding reciprocal influence. The majority of our work during the internship was done together, while some practical task have been done more independently: for example, Mathieu designed the scripts set about manual frequency settings (described in chapter 4) while the study on WRF and GROMACS application presented in chapter 7 was mainly performed by me.

1.1 Contents of the thesis

In chapter 2 we start by presenting the problem of energy savings in the HPC application domain, exposing the motivation behind our approach. First, we found the need for an energy-saving software solution that is independent of the target application implementation. Second, it has to reduce the power and energy consumption on the machine at runtime, with a minimum impact on the execution and minimum performance loss. This is possible only by monitoring the machine behavior during application execution by means of real-time metrics. We then analyze the current state of the art on the problem by comparing the related work with our approach: we will show that the main technique used to reduce the power consumption at runtime is the Dynamic Voltage Frequency Scaling (DVFS) technique. This technology allows changing the CPU operating frequency at runtime, reaching energy savings under certain conditions. We conclude by presenting our solution proposal, listings our initial objectives: reach real HPC application optimization, no source code modification in the target application, no performance degradation while saving energy, and no application profiling before optimization.

Chapter 3 presents the theory and all the concepts that are the basis of our solution. We first explain the relationship between the energy, power and execution time, in addition to measure unit used to express the energy. Then we pass to an overview of cluster energy consumption components, identifying main energy consumers: first among all the CPU, then the RAM memory, the Hard-Disks, and cooling system. Going further with the CPU, we present a power consumption model, that explains the central role of the

operating frequency for the power consumption of the CPU. We also describe all the means the CPU has to save energy, notably the C-states and P-states. Furthermore, we analyze the different HPC applications behaviors, finding the three majors: CPU-bound, memory-bound and balanced behavior.

In chapter 4 we describe our experimental environment and methodologies. Firstly, we describe the hardware and software technologies of the cluster we used for our development and tests. We present the architecture of the system and computing nodes, together with interconnect technologies and the Slurm workload manager. We that explain our experimental methodology notably with all steps we followed for hardware setup and measure process.

Chapter 5 contains the preliminary tests we performed on micro-benchmarks in order to study the machine behavior through hardware counters. We then describe actual counters and libraries we used to access them. Furthermore, we present our micro-benchmarks study with codelets analysis, expected values, and values retrieved through counters. We conclude the chapter with an interpretation of the results: we show how operating frequency and memory size impacts on performance and energy consumption.

Thanks to obtained results and theory bases, in chapter 6 we present the Bull Dynamic Power Optimizer (BDPO), the system we designed as main objective of this research project. We start briefly with the product company context, explaining where its idea comes from. Then, we pass to a general description of the BDPO design, describing the role of each module inside the tool. Furthermore, we explain the functioning for most important modules: the main Control Loop, the Metric module, the Action module, and the integrated Profiler. For the Metric module, in particular, we described in deep how we combined hardware counters to retrieve the Instruction Per Cycle metric (IPC). This metric was the one we used for all subsequent tests with real applications. Thanks to the IPC it is possible to detect different phases of computing stress on the CPU, so that it is possible to gain energy by reducing the operating frequency when it is not needed.

Chapter 7 shows then all BDPO tests we performed on real HPC application: after a presentation of performance metrics we used during tests, we briefly describe the three applications we used for our validation: GROMACS, WRF and NEMO. Then, we presented all the studies and results we obtained for each application. We tested those applications on homogeneous compute nodes, configuring executions from single node up to six nodes involved in the computation. From our study, we find out that GROMACS is a CPU-bound application, for which the maximal operating frequency is already the optimal one also for energy consumption. An IPC profile of GROMACS is then presented. For the WRF application, instead, we find a peculiar behavior of the IPC metric: the IPC profile of the application was highly variable, characteristic that prevented our tool to have a significant energy gain. In the WRF case, BDPO was conservative from the energy point of view (with respect to references cases), while it introduced around 5% of performance degradation. We then find out that for some application the IPC

metric, alone, could not be sufficient for phase detection. The NEMO application leads to more interesting results: BDPO managed to reach an energy gain from 8% to 11% compared to references cases, while it introduced an overhead of less than 1% in both cases.

We concluded our thesis with chapter 8, where we compared our initial objectives with the obtained results. We reached real HPC application energy saving, as demonstrated by results we just cited where we gained around 10% of energy consumption; BDPO is a tool that does not require to modify applications source code, as it works without any code annotation in the actual application implementation; we managed to have a slight performance degradation, especially taking into account the prototypical nature of BDPO; for the profiling of the application, we had to assume that the user has a sufficient knowledge of the application to configure BDPO. In this chapter we also discuss the improvement that we plan for the future of BDPO, explaining our perspectives after our study, implementation, and testing of this system.

2 Power consumption optimization of HPC applications

This chapter presents an overview of the problem of energy saving in the HPC applications domain. Then, we will focus on describing the current scientific state of the art, with the work already done related to this problem. We will try to highlight the similarities and dissimilarities of our approach related to all the other presented in the state of the art.

2.1 Overview

In the framework we presented in our introduction, we came with the question: how to reduce the energy consumption of an HPC application? First, it is very likely that people in charge of this model-to-code translation are mainly focused on problem solving and performance issues, and they don't care about the energy impact of their implementation. However, we will show later in this thesis that optimizing performance is also a key factor for energy saving. Second, programmers do neither have the needed knowledge about the subject of energy savings on HPC systems nor any information about the machine on which their code will run. Third, for all existing implementations, programmers would not like to change what they have already produced, so that is difficult that they would modify their code using a hypothetical third-party API for energy efficiency, besides huge libraries like the already cited. Fourth, people often don't want to perform static analysis of their applications before its actual execution in order to know in advance the best machine parameters for optimal energy consumption.

In this context, we found the need for an energy-saving software solution that is independent of the actual implementation of any real HPC application and that would reduce the power and energy consumption on the machine at runtime, with a minimum impact on the execution and performance loss.

2.2 State of the art

Speaking of software solutions for energy saving, the main technique used by different tools is the Dynamic Voltage Frequency Scaling (shortly DVFS) that, in a few words, is the possibility to choose and set the operating frequency on a CPU; we will detail this technology better in chapter 3. Moreover, in 3.3.1 we will explain why this technique allows energy saving and the relation between the latter and the CPU frequency. Triquenaux [3] identifies three

primary DVFS controllers: REST, UtoPeak, and FoREST-mn. Furthermore, Intel has recently released a paper on GEOPM [4], an open source runtime framework for energy optimization across compute nodes. Moreover, Unni et al. [5] propose an energy optimization approach for MPI applications relying on DVFS at node level applied by an optimization agent.

Before describing already cited solutions, it is necessary to cite also the Mont-Blanc project [6]. This is an European project started in October 2011 to design a new type of computer architecture capable of setting future global HPC standards, built from energy efficient ARM solutions initially used in embedded and mobile devices. Bull s.a.s has an important role in this European project, while from October 2015 it coordinates a new phase of the project. This phase adopts a co-design approach to ensure that hardware and system innovations are readily translated into benefits for HPC applications. It aims at designing a new high-end HPC platform that is able to deliver a new level of performance/energy ratio when executing real applications.

REST

REST [7] that stands for Runtime Energy Saving Technology is a purely on-the-fly DVFS controller that provides a software layer that uses hardware performance counters and gives users a plausible energy consumption improvement with their current hardware set-up. The basic idea behind REST is to select a frequency regarding an application phase behavior. First, it needs a way to measure the application activity. Then, based on that activity, a trend has to be selected as we will also explain in 3.4.1. Depending on which trend it is found, a frequency is selected. Finally, once a frequency is chosen, it has to be applied.

The work described in this thesis use an approach very similar to REST, even though the main purpose of the latter is to optimize energy on a single processor machine, while our objective is to target multi-cores HPC clusters running real HPC applications.

UtoPeak

UtoPeak [8] is an offline profiling tool, which analyzes the application and determines the best sequence of frequencies providing the lowest energy consumption for a given program execution. Having this sequence, it is possible to evaluate the energy reduction performed by a dynamic system such as REST. In fact, UtoPeak was designed to primarily evaluate the efficiency of REST: it computes the maximum reduction of energy consumption that is possible to expect from the use of a DVFS controller. Knowing with this technique the higher bound on energy reduction, it is possible to have a reference value to evaluate every DVFS controller.

As we said, UtoPeak performs an offline analysis of the application behavior, giving at the end the best parameters for a specific run of an application. In our work, on the contrary, we will use an opposite approach: we try

to explore the effectiveness of dynamic, on-the-fly optimization. The objective is to design a solution that is able, at runtime, to reduce the energy of an HPC application.

FoREST-mn

FoREST-mn [9] is the actual evolution of the simpler tool REST. It tries to solve some of the limitations inside REST, such as its naive decision making and the need of hardware counters specific to the Nehalem architecture. Additionally, FoREST-mn has the possibility to bound the performance loss of the tool itself to a given threshold. Furthermore, it tries to extend the single-node nature of REST, taking into account the communication slack typical of multi-nodes applications.

Even if FoREST-mn seems to be a really good DVFS controller with really nice energy savings, it needs to annotate the source code. As we explained in 2.1, one of the crucial points of our work is to be completely independent of the HPC application that needs to be optimized: we do not want to change anything inside the application source code. This will avoid many problems that potential users could find using the tool because they do not need to integrate the tool itself inside their applications. Then, we point to a solution "as simple as possible" that, starting from scratch, try to reach the maximal effectiveness with the minimal effort.

Intel GEOPM

In last years Intel disclosed the Global Extensible Open Power Manager [4], an open-source framework that point to have a centralized controller on exascale clusters, rather than a per-node monitoring. It leverages application-awareness to identify compute nodes on the critical path in an MPI job then diverts power from nodes outside of the critical path to accelerate the critical path nodes. Power and performance are adjusted via hardware counters and the DVFS technique.

MPI multi-agent optimizer

Unni et al. [5] propose an energy optimization approach for Message Passing Interface (MPI) applications running on HPC systems. Their approach is based on a Multi-agent based energy management framework, which uses an optimization agent for implementing energy optimization algorithm. This technique is applicable in two situations. First one is based on a master-slave model in which, usually when the master process is executing its task, all the slave processes are in waiting state. During this execution time, all of the slave processes are idle. Even though they are in waiting state, processors operate at high frequencies, wasting energy. Our idea is to reduce the processors frequencies on slave nodes as long as slave processes are idle, so that the power wastage can be minimized. Second situation is when processes

carry out the I/O operation, all the nodes on which these processes run, operate at higher frequencies with CPU low utilization. Hence, reducing power consumption during this time, power wastage can be minimized.

This approach has with us the common objective to be independent from application source code. It does not need any code annotation, even if it is applicable only on a certain domain of MPI applications (master-slave). However, we found during our studies (section 3.3.2) that activating C-state on compute nodes it is possible to reach important energy savings during idle phases and I/O operations.

2.3 Problem analysis and solution proposal

As we saw, reducing energy consumption in HPC applications can be done in many ways, mainly statically, offline or dynamically. Static analysis are source code analysis, that do not involve any program run. Offline and dynamic analysis are, respectively, studying the application behavior before its execution or acting directly on it during its actual execution, at runtime.

We will not face static analysis in this thesis, while we target to optimize applications energy consumption dynamically. As we saw with Utopeak in 2.2 offline analysis often require at least one complete run of the application in order to analyze it and find the best way to optimize the energy consumption for that job. As we presented in 2.1, many programmers and customers prefer to have an automatic way to optimize on-the-fly their job on an HPC cluster rather than pay the cost of pre-run analysis. For sure, for our solution, we will assume that the user has at least a good knowledge of the behavior of his application. For this assumption, we had to perform some offline analysis on target applications to understand how they behave before optimizing them.

We propose then a software solution that will monitor the behavior of each computing node through hardware counters, detecting the phase the application is currently facing and that will act on the system itself depending on collected data and phase detection, notably changing the operating frequency of the processor with the DVFS technology.

The main problem of having a solution working at runtime is the overhead introduced: obviously, the tool will need some resources while running together with the application, causing some performance loss. For sure, in HPC we want to minimize as much as possible performance degradation, also because, as we will present later in this thesis, optimizing the execution time of certain applications is a way to gain energy.

Summing up, our solution proposal point to have those four main features:

- Real HPC application optimization, that is to say not only testing the tool on synthetic benchmarks where we know what to expect but especially on real cases, with real parameters.

- No intrusiveness on the application. The tool must be able to optimize the consumption of any job submitted on a cluster, without source code modifications.
- No performance degradation, or at least as less as possible and the closest to zero; in the worst case, it has to be conservative.
- No profiling needed on the application to optimize. Eventually, we want to create a tool that will be able to be completely automatic, and it will auto-tune its actions at runtime.

As we will present in this thesis, we reached the first three points, while the last has been replaced with some assumptions and some procedures before the launching of the application with our tool are actually needed. Anyhow, we always kept this point in mind, and we designed the tool to be easily upgraded in that direction, besides having thought possible solutions to be integrated and that we left for future studies.

Having seen an overview on the actual state of the art on application energy reduction, next chapter will explain the theory at the base of our solution design.

3 Theory and concepts on energy consumption

This chapter presents the theory and all the concepts that are at behind the design of our solution. We first explain the relation between the energy, power and execution time. Then we give an overview on cluster energy consumption components, identifying main energy consumers. Furthermore, we present a CPU power consumption model, that explains why the operating frequency could be changed to reach energy reduction. We also describe all the means the CPU has to save energy, notably C-states and P-states. Moreover, we analyze here HPC applications behaviors: CPU-bound, memory-bound and balanced behavior.

3.1 Energy model

In order to express the consumption by a component or an entire system, we will use the most basic formula that relates the energy to the power and the time. The pure energy consumption is defined as:

$$E = P \times T \tag{3.1}$$

where the energy E is computed as the power consumption P of the studied system multiplied by T, the studied time period. With this simple formula, we can clearly see which are the two orthogonal levers at our disposal to module energy consumption, and it is also intuitive to understand that the total amount of energy consumed by a device is directly proportional to both the instantaneous consumption of the device (its power) and the period of time in which it will consume that power. In section 3.3.1 we will also see that the variations of P and T are quite opposite, meaning the the energy optimization of an HPC system is often a research of the best trade-off between optimizing the execution time and the consumed power.

The most common unit to express energy amount is the Joule[1] (J), but as the previous formula suggest, in electricity the energy is defined as watt-per-hour or watt-hour (Wh), where the relation between the two is

$$1Wh = 3.6 \times 10^3 J.$$

In this study we will use both of them, but in the majority of the cases we will refer to the more general unit, the Joule.

[1] In the SI the Joule is defined as $J = \frac{kg \cdot m^2}{s^2}$

3.2 Cluster energy consumption breakdown

As we already said, an HPC cluster contains many elements, all of them have different energy consumption. Moreover, each component consume a different quantity of energy depending on the context, the type and the amount of job to be executed. Between all components, we can try to identify which are the main energy consumers: the CPU and the motherboard, the RAM memory (Random-Access-Memory), the hard-disk-drives (HDD) and the cooling system. The tool presented in this thesis will act directly on the consumption on the CPU, that will be described in details in section 3.3, but it is important to know if and how it is possible to gain energy on the other components. In the following we briefly describe the consumption of HDD, memory and the cooling systems.

3.2.1 CPU

Still nowadays manufacturers are trying to respect the Moore's law: *"the number of components on an integrated circuit doubles every two years"* [10]. For sure, it exist a physical limit to this incredible growth, and we are every year closer to it: in 2014 Intel introduced the 14 nanometers CPU architectures [11]. Obviously we couldn't have respected the Moore's low by keep expanding a single processor, because of the Thermal Design Power (TDP): the increase reached a point where any standard cooling system, like fans presented above, were not able to keep the CPU in an acceptable temperature range.

After three generations of hardware design, Intel decided to re-design the Pentium M family for multi-core. It gave birth to the Core family processors. The new family was offering a TDP ranging between 10 to 150 Watt. The 150W was obtained on the extreme editions of the family which were quad-core processors and no longer single core as the Pentium D.

Besides the enhancement and addition done to the Core processor family, power management features were added. Under the name of Intel SpeedStep and TurboBoost, leverages were offered to the operating system to manage the CPU operating frequency and efficiently encounter a wide range of situations regarding power consumption. All the means offered to the OS to manage the CPU power consumption are presented below in 3.3.

Though a lot of enhancements were performed on CPU thermal dissipation and power consumption, it still is acknowledged as the main consumer in most of the configurations.

3.2.2 Memory

The energy consumption of the RAM modules are, in percentage, much less relevant comparing to the CPU consumption, described later on in this chapter. Nonetheless, when it comes to the energy consumption of a full HPC

cluster, they cannot be neglected, due to the high number of RAM units embedded in this kind of machines. As an example, we can take Sunway TaihuLight, the most powerful supercomputer in the world at the moment of the writing of this thesis, that has more the 1300 TB of RAM memory. If we consider that the most used RAM modules are those with 32 GB of memory, we can easily see that Sunway contains more than 40.000 RAM units, that is very likely to be close to the reality, view that its architecture is composed by little more than 40.000 nodes [1].

The RAM power consumption scales, in most of the cases, linearly with the voltage supply: based on Ohm's law we have $P = V \times I$, where P is the power, V is the voltage supply and I the current intensity. RAM modules have the possibility, through the BIOS, to choose their working frequency, but trying to lower too much the speed of the module could heavily impact the performance of the overall application. In fact, it is well known that memory access operations are much slower compared to the processor speed, meaning that very likely the memory become a bottleneck. A big drawback of manually changing the RAM operating frequency is that it has to be performed inside the BIOS. In other words, it means that the machine have to be rebooted. This operation is usually not possible in an HPC environment, because of the incredible loss of performances.

3.2.3 Hard-Disks and SSD

For Hard-Drives, the same type of reasoning we used for the RAM in terms of quantity can be used: modern HPC systems have an huge amount of mass-memory storage. For instance Titan, the fourth supercomputer in the TOP500 list [1], has 40 Petabyte of storage space [12]. Supposing the usage of 4 TB disks, the system will use a total of 10.000 disk modules. This number could be way less negligible in terms of energy consumption than what we can think about our personal computers: the energy consumption of 10.000 disk modules has a much higher impact on the overall consumption then one single Hard-Drive in our PCs.

Hard-Disks power consumption can be attributed to two main actors: mechanical parts and an electronic controlling device. The main idea behind its optimization is to stop working the mechanical parts, that is always the major consumer, when they are not needed. This strategy could be applicable at different levels : using a "standby mode" in which only the mechanical parts are down, or even deeper saving-energy states, till the total shutdown of the unit. With a different granularity, it is possible to tune the speed of a disk in relation to arrival rate of requests. The drawback of this strategy is that the deeper the sleeping state, the higher the amount of time needed to reach the nominal speed, causing extra latency and, thus, performance issues.

In the last years, Solid State Drives (SSD) are becoming more and more attractive from both performance and energy consumption point of view: SSDs have no moving mechanical components, using instead flash memory. This is the main point of SSDs with respect to HDDs: laptop hard disks, for example, consume about 2 watts while in operation. In contrast, the power

consumption of flash SSDs during operation is in the range of few hundred milliwatts (mW), and it consumes only a couple of micro-watts (μW) when idle [13]. Surely, the main drawback of SSDs is the cost per GB, being much more costly that traditional HDDs. We don't have recent and precise price comparison, but in last years the gap is continuously reducing. For their performance, robustness, and low power consumption, storage systems using flash memory are then rapidly becoming popular in diverse computing systems.

3.2.4 Cooling system

When speaking about cooling solutions for computing systems, it is really important to think about the size scale of the system taken into account. In HPC systems, a lot of the power injected into the electrical components, especially CPUs, is transformed into heat. Thinking that HPC computers have a power consumption in the order of magnitude of Megawatts, the produced heat amount and that has to be managed is huge. Actually, taking the total energy consumed by a typical air-cooled cluster, on average 50% of that energy is not used by the computing process itself, but by powering the necessary cooling systems [14] [15]. This data shows how crucial is efficiency in cooling systems in order to save energy for a whole HPC cluster.

Recently, a lot of HPC centers are switching from air-cooled to water-cooled systems, where the latter allow a much higher efficiency and reduced costs in term of energy. In [15] R. Mahdavi shows that for the Maui High Performance Computing Center (MHPCC) a much less cooling power is required by the water based system, compared to the cooling power required by the air cooled one. From their testing, they estimate that the new solution will save 200,000$ per year in operating costs.

3.3 CPU power management

During last decades CPUs had a swift, non-stop evolution, regarding size, the number of transistors per area unity, compute power, the number of processing unit and also energy efficiency. Even if many enhancements were performed on CPU thermal and power dissipation, it is still acknowledged as the main consumer in most configurations. During years, more and more leverage was offered from the hardware to the OS to manage the CPU power consumption. This section presents all the basic concepts and the means used in our work.

3.3.1 Model of the CPU consumption

Before acting on CPU parameters it is important to understand how the CPU consume its energy and dissipate its power, identifying the main actors that

contribute in this process. According to many publications, the most common model for CPU power consumption is as follows:

$$P_{CPU} = P_{dynamic} + P_{static}$$

where $P_{dynamic}$ is mainly induced by the activity of the CPU, the higher its usage the higher its consumption is. On the contrary, the P_{static} term is due to hardware imperfections such as leakages and wire capacitance. We also know that the P_{static} changes linearly with the temperature because of the power leakage of the silicon die, but we will consider it as constant and invariant in comparison with the $P_{dynamic}$.

Going more in detail in the definition of the $P_{dynamic}$ term, it varies regarding the CPU activity as shown here:

$$P_{dynamic} = A \times C \times V^2 \times f. \tag{3.2}$$

The activity factor A quantifies the percentage of active gates on a processor. For example, if a parallel application is using all cores available on a processor, the value of the A factor will be higher than a sequential program running on only one core. C is the sum of all gates capacitance, while V and f are respectively the voltage supply and the operating frequency. As it can be noticed from the latter formula, $P_{dynamic}$ is quadratic to V, so that lowering the voltage means important power saving. However, voltage is not the only factor to be taken into account in the energy consumption, all the more so as it does not vary independently as a function of the other factors, notably the frequency f as described later on: we will see that the operating system sees the operating frequency and voltage as a couple called P-state. Moreover, we know that there exist a direct relation between operating frequency and the system temperature, so that lower the frequency causes a reduction of the P_{static}, that as we said changes linearly with temperature. In addition, as it will be explained in the next paragraph, as P-state above, processors expose additional states, where sub-parts of the system are shutdown. Parts that are not powered cannot leak, inducing more power saving on P_{static}.

3.3.2 ACPI and OSPM states

As we said before, the OS has a set of leverage in order to control the overall CPU power consumption. The ACPI (Advanced Configuration and Power Interface) is an open industry specification co-developed by Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba [16]. It defines a platform independent interface for configuration and power monitoring and management. It allows the users to control a lot of parameters of the whole system, for example CPU operating frequency, fan speed or to monitor CPU temperature.

In order to manage the power consumption of the CPU, besides exposing hardware features through dedicate API, the ACPI allows Operating System directed Power Management (OSPM): the OS defines some "states of consumption". For example, if the OS figured out that the machine is without

job for a certain period of time, there is not need to operate at full speed. Using this strategy, the Operating System can manage its power by modifying the resource behavior depending on the state it is in. There exist states at different granularity: global states, processor states and performance states.

For the whole, global system G0 is the working state, while G1 corresponds to the inactive or sleep state and G2 is the shutdown state. Then, while the system is active (G0 state) the CPU faces different utilization: processor states (C-states) ranges from C0 to C7, where only C0 is the working state and all the other are deeper and deeper idle states, each of them implying higher power savings. It is also important to do not mistake between idle states, that concern only the CPU, with sleep states, that are proper of the global system.

Idle states from C1 to C7 have various entry/exit-state latency: as an example, for Intel® processors [17] the C1 states corresponds to halted core while cache coherence is maintained. At C3 the L1 and L2 cache content is flushed into the L3, shared cache. In deeper C6-C7, depending on the model, the core architectural state is saved in RAM and the power supply of the core is shutdown. As one can imagine, the deeper the C-state, the higher the cost in term of time needed to return at C0, the active state.

When the core operates at C0 multiple performance states, called P-states, exist. Whose number depends on the processor model: at each P-state is associated both an operating frequency and a voltage: this is due to the fact that voltage and frequency are strongly related, according to the transistor physics. Lower voltage implies slower transistor commutation, inducing lower operating frequency. As we saw for C-states, also for P-states exist a delay in changing between states: in fact, the voltage transitions between P-states is not instantaneous since voltage regulators are controlled systems [18]. If the processor asks to scale the frequency in an ascending shift, meaning switching to higher frequencies, the voltage regulator has to scale up to meet the required tension according to the selected frequency. If it is a descending shift, the frequency can be immediately switched without waiting for the voltage to reach the correct level. Indeed, on the first hand, there exists latency between the request for a frequency switch, and the moment it is effective. For instance, for the Ivy Bridge micro architecture, scaling frequency from 1.6 GHz to 3.4 GHz is associated with a latency which is around 45 microseconds [3], including the fact that an operating frequency scaling implies a full flush of the CPU execution pipeline. Therefore, the commutation time of pair voltage/frequency cannot be neglected.

As before, C-states and P-states should not be confused with each other. C-states are idle power saving states, in contrast to P-states, which are execution power saving states. During a P-state, the processor is still executing instructions, whereas during a C-state (other than C0), the processor is idle, meaning that nothing is executing.

3.4 Energy profiling

To better understand the energy consumption of computation systems in relation to the type of requested job, a profiling study is needed. The objectives are to understand how and how much an application consumes, and we will see that it is possible to categorize applications in three main groups: CPU-bound, memory-bound and balanced behaviors. According to the type of application, it is then possible to perform DVFS, as described in 3.5. In a few words, it consists in setting the operating frequency to the value for which the energy consumption associated with the execution of the job is minimized. In chapter 5 some results based on those concepts will be presented.

3.4.1 Application trends

An application is CPU-bounded if its workload consists in computations. As a consequence, the higher is the operating frequency of the CPU, the faster the code execution, as much as pure integer and floating-point operations are concerned. As a result, it will lower the overall energy consumed since the decrease of the execution time will counterbalance the increase of $P_{dynamic}$ due to higher frequency (accordingly to equation 3.1 $E = P \times T$). As a general principle for this type of application, the highest frequency is preferred: it is a "run to finish" (except for Turbo Boost, when the consumption is too high to compensate the reduced execution time).

For memory-bounded jobs, instead, the main load is on data exchange between the CPU and the main memory (LLC and RAM). In fact, if the application is memory intensive, a huge amount of requests are going to be issued to the memory controller. However, accessing data stored in RAM is incredibly slow, compared to the duration of a CPU cycle. Thus, it results in a stall of core pipeline due to high latency of memory operations, which entails a bottleneck for the execution time. As a result, no matter the CPU operating frequency, the execution time of the application will remain the (almost) the same. This implies that the lowest operating frequency is to be preferred, since it leads to a much lower energy consumption of the overall application execution (decrease both $P_{dynamic}$ and P_{static}).

As expected, applications with a balanced behavior consists in an alternation of CPU-bound and memory-bound phases. The resulting energetic profile is not as straightforward as the two previously presented: for lower CPU frequencies, a balanced application behaves just as if it were CPU-bounded, so that increasing the operating frequency results in a decrease of the energy consumption; but after a certain threshold, its behavior changes and it becomes memory-bounded, so that the CPU is running faster than needed, regarding the memory access. This means that a minimal energy consumption does exist, for an operating frequency located somewhere around the threshold previously mentioned.

In addition, during the execution of an HPC application also I/O and communication phases exist. In a few words, an I/O phase is when a core has to perform a great amount of data writing and/or reading on disk. A

communication phase, instead, is when a process on a core is sending or receiving a message from another process, for data exchange or synchronization purpose. In both described cases, the core computational power is not needed because the execution pipeline is waiting the completion of data exchange operations. In such a situation, the core will try to enter a deeper C-state, in order to save energy as described in 3.3.2. We will better describe I/O phases with the NEMO application in chapter 7.

3.4.2 Phase detection

The previous section introduced balanced-bounded applications, and notably the fact that they consists in an alternation of CPU-bounded and memory-bounded phases. In order to study this alternation, two main methodologies are possible: static and dynamic phase detection.

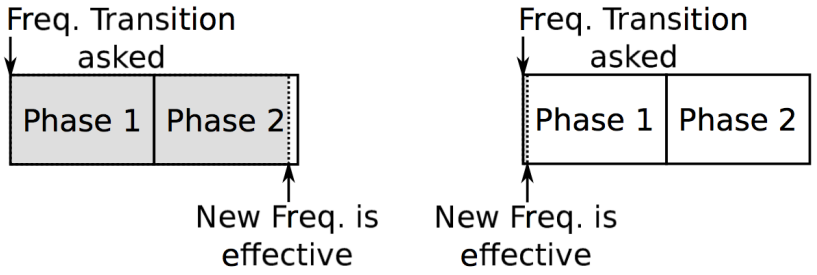
Static phase detection consists in analyzing the source code of the requested job and extracting some codelets, that should be representative for the most significant part of the entire application execution. Then, a single-codelet analysis is performed in order to understand the behavior of each codelet. Finally, by putting together all codelets results, an overall behavior prediction is given. Then, according to this prediction, the optimal operating frequency can be selected.

Dynamic phase detection uses a different approach to retrieve phases information: during a real execution of the application, several metrics describing the functioning of the system are monitored on the fly. From the measurement, it is possible to link a particular execution phase of the application to a specific set of values of those metrics. It will thus be possible to dynamically, i.e. at runtime, determine the current type of execution phase of the application, and adjust the operating frequency accordingly to minimize the energy consumption.

The metrics in question typically are the number of instruction per cycle (to have a picture of the CPU stress) also called IPC, and the number of cache misses (to have a picture of the memory stress). Those metrics are often a mathematical combination of exposed hardware counters, as we will explain in sections 5.1 and 6.4.

3.5 DVFS

As it was seen previously, on one hand, the OS has different level of states in order to manage its power consumption, and on the other hand, we described the various natures and possible phases an application may have. In order to connect those two different notions, the DVFS concept is built. The idea behind DVFS is to dynamically adapt at runtime the P-state depending on application phases, to reduce power consumption and hopefully the energy. This means that during the execution of the target application we ask to the OS to change the CPU P-state, trying to choose the P-state that fit the best the current phase of the application, so that the energy consumption associated with the execution of the job is minimized.



(A) Phase too small for latency (B) Good phase size for latency

FIGURE 3.1: Frequency transition latency vs phase duration [3]

It must be said that applying DVFS techniques to a system may raise some issues. To begin with, even if the main goal of DVFS is to lower energy consumption of a machine, it must not be forgotten that it may downgrade its global performances. When it comes to exascale HPC systems, performances are just as important as energy consumption. Thus, a trade-off has to be found between the latter, and DVFS should be parsimoniously implemented. On top of that, DVFS implies scaling the operating frequencies of the CPU. However, as it was seen in previous sections, performing such a shift is not cost less.

This proposition leads to a logical conclusion: since there is a real latency for the frequency scaling, it doesn't make any sense to switch the frequency for phases that have a duration comparable to the latency, as presented in Figure 3.1. The case where a phase duration is comparable to the transition latency is shown in fig 3.1a, where the effective frequency change happens too late, even worst in the following phase. Conversely, fig 3.1b presents the good case, where the first phase can take advantage from the frequency choose for its state. Estimate the time duration of a phase in order to determine whether or not a frequency scaling will be beneficial is one of the most complex task when it comes to implement DVFS techniques.

Before passing to our micro-benchmarks study, in next chapter we will describe the experimental environment and methodologies we used for all our study, development, test and validation.

4 Experimental environment

This chapter contains a detailed description of all hardware and software supplies we used during our internship. In particular, we will describe the cluster that we used for development, testing and measuring purpose. Moreover, in this chapter we will present the experimental methodology and scientific approach we followed for all our work.

4.1 Hardware and software environment

In the Research and Development department of Bull we had the possibility to develop and test our product on some compute nodes of a real HPC system. The following is a description of the machines we used, from both the hardware and software point of view.

4.1.1 The cluster

All our development, tests and measures have been done on a Bull internal development HPC cluster, called MEXICO. This cluster is a CPU-based cluster, that is to say that is designed to execute parallel applications thanks to the multi-core architecture of Intel® processors. CPU-based clusters differs from GPU-based clusters mainly for number of cores and computing power of each core.

The cluster contains more than 60 computing nodes. Being a development cluster, it have to serve a lot of different purpose, so all nodes are not homogeneous, notably by number of cores and memory size. The cluster is also shared by many development team, so it is divided in logical partitions, that can be reserved thanks to the Slurm workload manager, that we will detailed better later in section 4.1.2.

All the nodes in the cluster are connected through both Ethernet and the InfiniBand interconnect technology. Ethernet is used mainly for administrative operations on each node, like ssh user connections, while the much faster InfiniBand connections are used for message and data exchange between nodes during computing operations.

On top of that, the MEXICO cluster is organized as follow: some of its nodes are so called login nodes, that acts as entry points for all connections, notably through ssh. Than, all the nodes mount also a Network File System (NFS). This network protocol permits to use the network between nodes to access remote hard drives as local hard disks. With this file system a user account placed on a server node can be shared among all nodes, having all same files synchronized and available on all connected machine.

NFS is useful for administrative issues and users managing, but is not optimal for HPC computation. During the execution of an HPC application, in fact, a lot of different processed would try to write at the same time on the same disk. This could leads to performance issue if not managed in a proper way. In Bull, Lustre file system is used to manage HPC input/output operations. Lustre is a distributed file system, optimized for large-scale cluster computing. The MEXICO cluster has then some dedicated nodes for I/O.

During our internship we worked with a part of the MEXICO cluster, using a total of ten different nodes. Those were logically divisible into two groups, homogeneous in themselves. Four nodes mounts an Intel® Xeon® CPU E5-2650 v3 with a nominal frequency of 2.3 GHz and 20 physical cores (40 logical with Hyper-Threading activated). The other 6 nodes have an Intel® Xeon® CPU E5-2670 v3, nominal frequency 2.3 GHz, 24 physical cores per CPU (48 logical cores with Hyper-Threading). All nodes CPUs belong to the same hardware architecture family of Haswell processors, and all of them mount the same Linux Operating System: Red Hat.

4.1.2 The workload manager

As we said, heaving a big cluster means to manage it. In our case, multiple teams had to work on the same development system, while in a production environment maybe a cluster owner have to serve multiple customers with different job requests. This means that, in a development environment, two teams have to work compulsorily on disjoint set of nodes, otherwise it is very likely that they will invalidate the measures done from the other team, interrupt their work an so on. An effective system must be used, especially when the size of a company, a project or the number of customer is too high to be managed by a straightforward direct agreement. In Bull, we used the Slurm workload manager [19].

Slurm is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters. Slurm requires no kernel modifications for its operation and is relatively self-contained. As a cluster workload manager, Slurm has three key functions. First, it allocates exclusive and/or non-exclusive access to resources, usually compute nodes, to users for some duration of time so they can perform work. Second, it provides a framework for starting, executing, and monitoring work, normally a parallel application, on the set of allocated nodes. Finally, it arbitrates contention for resources by managing a queue of pending work.

During our internship we used all those three features: we created a reservation for our test and measures that we will describe in chapter 7; we managed the ten nodes we had into two logical groups, dividing them by different number of cores; we monitored the applications we launched and we retrieved job duration and job energy consumption thanks to the Slurm manager.

4.2 Experimental methodology

In this section we will explain the experimental methodology we followed in order to make sure that the state, set-up and experimental conditions were the same on all nodes we used. Moreover, we will present the strategy and mathematical means we used for all measures performed in our study.

4.2.1 Experimental hardware setup

Here, the "hardware setup" refers to the configuration of each node used for our test, mainly regarding the setting of the frequency driver of the CPU. We here explain all the step we followed to set the appropriate parameters for C-states, hardware counters, frequency driver and thread pinning.

C-states

First we ensured that on all nodes C-states were active. If C-states are disabled, each core will never goes in a deeper state than C0. This means that it is always in an active state. In order to activate at least the C1 state, we had to set the kernel parameter `idle=halt`. By doing this, we ensure that each core will go at least into C1 state, where it performs only halt operations.

Hardware counters

We will explain in chapter 5.2 and 6 that hardware counters are a key point of this study. In order to read counters values we used the `perf_event` interface included in the Linux kernel. It notably aims at retrieving the values exposed by the hardware performance counters of the CPU. It is customizable, especially through flags exposed by virtual files. One of those flags is named `perf_event_paranoid`, exposed by a virtual file. This flag can be set in the range from -1 to 2: the higher the value, the lower the number of retrievable counters. It also controls the permissions level required to access the counters. We set the value inside the virtual file to -1, in order to have entire access to counters. We will detail more the `perf_event` module in 5.1.

Frequency driver and governor

There exist several different CPU frequency drivers, among which *Intel_P-State* and `acpi-cpufreq`. In order to use DVFS manually to dynamically choose the operating frequency of the cores of the CPU we needed to set the second one as frequency driver on our nodes.

The `acpi-cpufreq` driver utilizes a part of the ACPI interface dedicated to P-states control [20]. For this ACPI driver, besides many configuration parameters, several frequency governors are available and one of them allows frequency setting. A frequency governor is a way to define the OS strategy to manage frequency scaling. `acpi-cpufreq` notably has the following governors [21]:

- `ondemand`: it sets the CPU frequency depending on the current system load. Load estimation is managed by the scheduler and exposed to user-level space; when triggered, the governor checks the CPU usage statistics over the last period and the driver sets the CPU accordingly.
- `performance`: this governor sets the CPU statically to the highest available frequency.
- `powersave`: it fixes the CPU operating frequency statically to the lowest one.
- `userspace`: this governor allows the user, or any userspace program running with root privileges, to set the CPU to a specific frequency by writing a virtual file `scaling_setspeed` available in the CPU directory.

Our test machines were set by administrators to use *Intel_P-State*, which is loaded by the Linux Kernel at boot time. Consequently, first step consists in disabling the *Intel_P-State* driver. To do so, it is required to add a kernel flag and reboot the machine. Then, in order to select the desired frequency governor, it is necessary to write in a virtual file, checked by the OS frequency driver, the name of the desired governor.

View that for our experiments we had to change a lot of times both the governor and the operating frequency on a set of nodes, we designed a set of utility scripts written in bash. Those scripts allow to easily choose the governor on a set of nodes, and for the `userspace` governor, to set the operating frequency.

Cores management and Hyper-Threading

For all our experiments we had to ensure that each parallel process was located in the correct CPU core. Intel® processors also embed the Hyper-Threading technology, that permits to expose two execution contexts on a single physical core. This technology is powerful, but it may lead to higher results variability. To enhance the reproducibility of our results we decided then to avoid the usage of Hyper-Threading, by means of thread pinning through the `taskset` Linux utility and choosing an appropriate number of processes to be placed on one node by the MPI library.

4.2.2 Measurement methodologies

During our internship, we faced many different use cases, for which was impossible to apply always the same experimental protocol, especially in terms of number of measures. Anyway, for all the experiments we will describe in chapter 5 and 7 we always used the technique of experiment repetitions, performing our test multiple times. The objective was to ensure as much as possible the reproducibility of our results. Moreover, for tests with lower

number of repetitions we used the *median* mathematical tool, to exclude possible outliers. Instead, for tests with an high number of repetitions we preferred the *average* tool, in order to take into account all the measured cases.

5 Study of micro-benchmarks with hardware counters

In order to monitor the behavior of a compute node at runtime, we need a mean to retrieve some numerical and quantifiable metrics, notably some specific performance monitoring events. This section presents a brief description of the libraries used in order to read hardware counters. All those libraries stand on top of the `perf_events` interface, a software layer that allows the access to counters present on most processors. Then we will describes the actual counters used in our study and the micro-benchmark study we did related to those counters.

5.1 Hardware events and counters

Performance monitoring events are typically provided by the hardware or the OS kernel. The most common hardware events are provided by the Performance Monitoring Unit (PMU) of modern processors. They can measure or counts events related to the hardware, like elapsed cycles, the number of cache misses or the number of executed instructions from a certain point on. Then, there exist also software events, that usually count kernel events such as the number of context switches, or pages faults.

Programming events is usually done through a kernel API, such as Oprofile, perfmon, perfctr, or `perf_events` on Linux [22]. The `perf_events` interface is a Linux API added into the Linux kernel in version 2.6.31, and it was originally called Performance Counters for Linux (PCL). This API is a low-level instruments to reach performance counters, notably exposing together both the information from PMU and from kernel counters. A lot of contributors have used `perf_events` as a base for higher level libraries, in which the best known are libpfm [22] and PAPI [23] [24].

libpfm

The goal for the libpfm project is to develop a user library to help setup performance events for use with the `perf_events` Linux kernel interface. The idea behind this library is to simplify the procedure of events selection and monitoring, while adding some models and metrics on top of `perf_events`.

PAPI

The Performance API (PAPI) project specifies a standard application programming interface (API) for accessing hardware performance counters available on most modern microprocessors. This library expands and integrates the libpfm library, providing a consistent interface and methodology for use of the performance counter hardware found in most major microprocessors. PAPI supply two interfaces to the underlying counter hardware; a simple, high level interface for the acquisition of simple measurements and a fully programmable, low level interface directed towards users with more sophisticated needs. PAPI defines also some proper counters, allowing the access to high level monitoring like network and file systems counters.

5.1.1 Counters

In order to better present the micro-benchmarks study proposed in 5.2 and other studies in this work, we describe here the specific counters we used as metrics and their meanings. We recall that the presented counters are currently available for Intel® Haswell micro-architecture.

Instruction and cycles

One of the the main metrics that is really interesting to monitor and that we will show to be crucial in understanding the behavior of an application is the rate of instruction executed by the core. It is possible to measure this quantity taking into account the number of instruction fetched by the core pipeline and the number of clock cycles needed to execute them.

`INSTRUCTION_RETIRED` The instruction retired counter add one each time an assembly instruction is completely executed by the core pipeline, that is to say, the instruction is retired from the pipeline. For instructions that consists of multiple micro-ops, the counter counts the retirement of the last micro-ops of the instruction.

`READ_TIME_STAMP_COUNTER` The Read-Time-Stamp-Counter (RDTSC) is an assembly instruction that can be used to retrieve the count of the number of ticks since the last system reboot as a 64-bit value. It is, in a few words, a counter usable for knowing a period of time, usually expressed in number of clock ticks or clock cycles. The frequency of counting is the nominal frequency of the CPU.

`UNHALTED_REFERENCE_CYCLES` It counts the reference clock cycles while the clock signal on the specific core is running, that is to say, when the core is not halted. The reference clock, like for the RDTSC, operates at a fixed frequency (the nominal one), irrespective of core frequency changes due to performance state transitions. The main difference with respect to the RDTSC is

that the latter counts always, independently to core activity, while the URC only for unhalted instructions.

`UNHALTED_CORE_CYCLES` This counter is very similar to the URC, with the only difference that its counting frequency is not the nominal one but the actual core operating frequency. As an example, having a CPU with nominal frequency 2.4 GHz and an operating frequency of 1.2 GHz for the entire selected period of time, the URC will counts the double of the UCC.

Memory usage

Another important metric used to analyze the execution of an application is the behavior of cache and RAM memory usage. With the counters presented below it is possible to keep track of memory stress. We present here general counters for LOAD and STORE counting and low level cache monitoring.

`MEM_UOPS_RETIRED:ALL_LOADS` This counter take into account all memory load retired by the core pipeline. We recall that a LOAD assembly instruction is the request by the program to load the content of a memory address into an internal core register.

`MEM_UOPS_RETIRED:ALL_STORES` As for loads, this counts the number of stores instruction retired by the pipeline. A STORE assembly instruction is the opposite of the LOAD, because it is the request to store the value of an internal register into a memory address.

`L1D:REPLACEMENT` In the different levels of cache, it is possible to counts the number of cache line replaced. With this counter it is possible to monitor the number of level one data cache line replacement. It is very likely that a cache line replacement is triggered by a "miss" into the cache: another cache line will be searched in higher level of cache and it will replace one line into the current level of cache.

`L2_TRANS:L1D_WB` In the context of cache line transfers between levels of cache, is it possible to monitor specific transactions: this counter allows to count the L1 data write backs that access the L2 cache.

`L2_RQSTS:MISS` After a cache miss, when a lower level of cache ask for a new cache line to an higher level cache, the requested line could be present or not: this counter keep track of all requests that miss the L2 cache. Those miss will also cause cache line requests to the L3 cache.

Package energy consumption

The Running Average Power Limit (RAPL) [25] provides access to a set of counters for energy and power consumption information. RAPL is not an

analog power meter, but rather uses an internal power model. This model estimates energy usage by means of voltage, temperature, number of actives cores and internal performance counters. Both the counters below belong to the RAPL interface.

`rapl::RAPL_ENERGY_PKG` This counter is related to the energy consumption of the package, also called socket. For example, on nodes described in 4.1.1 there are two socket per CPU, each of them containing an half of the cores available per CPU. As we said, each socket has its part of RAM memory attached to, that in this case is not taken into account for this counter, while is measured by the following one.

`rapl::RAPL_ENERGY_DRAM` This counter provides a measure of the energy consumed by the RAM memory, and it is possible to differentiate the consumption, as explained before, between the RAM modules present in the two socket of the CPU.

5.2 Micro-benchmarks study

The characterization of hardware behavior studying hardware counters values on very small codelet of assembly instructions, that we call micro-benchmarks, give us the possibility to perform an important evaluation: we can count manually the expected values that we should expect from measured counters, and compare those with actual counters values retrieved. This process helped us to validate counters information.

We will also vary the data size of the micro-benchmarks to stress the memory hierarchy. We will set different operating frequencies on the core in order to study the energy consumption and the performances of codelets with respect of core frequency and data size.

5.2.1 Structure and parameters

In order to retrieve counters values for a codelet, we create a simple C program that is able to start the counters, launch the codelet, stop the counters and print the results. In listing 5.1 we have a simplified pseudo code example of the codelet call. As we can see, the two main parameters are the number of elements and the codelet repetitions.

```
1  int main() {
2      // Take number of elements and repetitions
3      read_inputs(nb_elements, repetitions);
4
5      // Compute memory size and allocate memory
6      size = nb_elements * sizeof(double) + 64;
7      allocate_memory(mem, size);
8
9      // Start monitoring
10     evaluation_start(counters);
```

```
11
12      // Start the codelet to be evaluated
13      __codelet(nb_elements, mem, repetitions);
14
15      // Stop the monitoring
16      evaluation_stop(counters);
17
18      // Print the values of the counters
19      print_results(counters, repetitions);
20
21      free_memory(mem);
22      return 0;
23  }
```

LISTING 5.1: Pseudo code for codelet call

The **number of elements** is the number of `double` data type elements that the main program allocates and that represent the amount of data the codelet has to deal with. By changing this parameters, considering that we know the hardware on which the codelet will be executed, we can forecast at which level of cache the data needed by the core will be placed. In our case, for example, we have a cache architecture as shown in table 5.1. If we want that our codelet fetches data out of the L1 cache, we have to ensure to allocate less than 32 kilobyte (the size of L1 cache). If we want, instead, to fore the working set to reside in higher cache levels, we have to allocate more than the size of cache L1. Let us assume, for example, that we want to fill the L3 cache memory: having a size of 30720 kilobyte, we can divide it by 8, that is the size of an element. We will find out that we need at least 3 million and 840 thousands elements. Obviously, enlarging the amount of data to work on, we will affect the necessary execution time, view that the codelet will have to work on more data. With this strategy, we can study the effects in term of energy consumption, power consumption and performance of two main factors: the number of cache misses and the latency on different levels of cache.

The **number or repetitions** is a necessary parameter for results reproducibility and stability: it simply is the number of times the codelet will re-execute its work. In fact, a single codelet execution could lead to variable counter values in between different runs. It is possible then to use a codelet repetition and take the average of the results. Moreover, as we just said, different memory sizes to work with means very different execution times: this could cause too short or too long codelet executions. So, we can tune the

Memory Level	Size
cache L1 (private)	32 KB
cache L2 (private)	256 KB
cache L3 (shared)	30720 KB
RAM	64 GB

TABLE 5.1: Cache and memory size on test node

Repetitions	Number of elements	Memory allocated
4 M	1 K	7.8 KB
1.3 M	3 K	23.5 KB
370 K	11 K	86 KB
160 K	25 K	195 KB
4 K	1 M	7.8 MB
1.3 K	3 M	23.4 MB
666	6 M	47 MB
333	12 M	94 MB

TABLE 5.2: Set of configurations for codelet study

number of repetitions in order to have a comparable total execution time.

The functions `evaluation_start()` and `evaluation_stop()` are those in charge to initialize, start and stop the counters monitoring. For our experiments we performed our measurements using the Performance API library (PAPI), already described in section 5.1.

As a strategy, knowing the cache and RAM sizes like described in table 5.1, we decided to choose two different data size per level of cache: one data size that is "small" for the cache size, and another that almost fill the cache level selected. With this procedure, we selected the combination of parameters showed in table 5.2. Each line of the table represent a configuration used to launch the micro-benchmark: it is possible to notice, as said before, that for lower number of elements are needed a very high number of repetitions. The execution time, otherwise, it will be really short. We found this parameters empirically, with the objective to have an execution time of at least 2-3 seconds. The table also shows for each number of elements the corresponding size into the memory. With this set of parameters we know that the first two rows will work on L1 cache, the third and fourth on L2 cache and so on, while the last two lines will need to access data in RAM memory, that is the the most common case for real HPC applications. It is important to say that the last configuration does not fill the 64 GB of available RAM, but simply goes a little further than the configuration just before. We made this choice to avoid a too long execution time, required by the codelet to deal with 64 GB of data.

The function `print_results(counters, repetitions)` take the results of counters and normalize them on the number of repetitions and number of loops needed to fit the requested memory size, permitting to evaluate the counters values as they were on a single execution of the codelet, presented in next section.

5.2.2 Codelet analysis

The herein below code listing 5.2 presents one of the codelets we used in our experiments:

```
1 void __codelet(long nb_elem, double *a, long rep)
2 {
```

```
3       long i = 0, j = 0;
4       do {
5           i=0;
6           do { // LOAD a[i]; Loads using 64 bytes strided accesses
7               __asm__ __volatile__(
8                   "movaps  (%[a], %[i], 8), %%xmm1\n\t"
9                   "movaps  64(%[a], %[i], 8), %%xmm2\n\t"
10                  "movaps  128(%[a], %[i], 8), %%xmm3\n\t"
11                  "movaps  192(%[a], %[i], 8), %%xmm4\n\t"
12                  :
13                  : [a] "r" (a), [i] "r" (i)
14                  :
15              );
16              i += 32;
17          } while( i < nb_elem );
18      } while( j++ < rep );
19      return;
20  }
```

LISTING 5.2: Codelet code

We recall that this codelet is called by the main process presented in listing 5.1, right after the starting of the counters and right before the counters stop.

The main purpose of this codelet is to force with each assembly instruction a new cache line to be requested from memory. As described in 4.1.1 each cache line has a size of 64 bytes. The idea is to load a couple of elements of the array in an internal register in order to force the load of the corresponding cache line into L1. Then, with a stride of 64 bytes, we repeat the process, ensuring that at each time a new cache line is requested.

To achieve this, the codelet is composed by four assembly instructions, encapsulated into two loops: the outer loop is the one that ensure to repeat the innermost loop as many time as requested by the repetitions parameter *rep*, while the inner loop is in charge to call the assembly instructions correctly on the entire allocated array. In a few words, the four internal instructions cover a segment of 256 bytes into the array, that corresponds to 32 double data (8 byte each one). Consequently, at each step of the internal loop, we increase the *i* elements counter of 32 unit.

For our experiments we pinned this codelet on a single core, that is to say that we ensure that the OS will not perform a migration of our execution on another core. With this method, we try to have a measure as clean as possible.

We recall that the results we expected from this experiment will be on average on all repetitions performed, on the total number of loop performed into the innermost loop. This means that the values obtained from the measures will represent the counters values for the execution of a single cycle in the innermost loop. Table 5.3 shows counters' results for the described codelet.

As we can see, the configuration is the third we presented in table 5.2, with a total data size of 86 kilobytes and 370.000 repetitions. We set the operating frequency at 1.2 GHz so that we expect a value for the `UNHALTED_CORE_CYCLES` counter almost the half of the `UNHALTED_REFERENCE_CYCLES`, having a dependency the former on the operating frequency and the latter on the

nominal frequency, that is 2.3 GHz, almost the double of the chosen one for this experiment. This expectation is confirmed by results, that as we can see shows an `RDTSC` and `URC` almost identical (no halted cycles are present during the experiments) with a value of 17.5 cycles, while the `UCC` shows 9.1 cycles, that is almost in the same proportions as the frequencies, as anticipated. With the described configuration we also expect that the L1 size will not be enough to contain the requested memory size. This will cause a continuous cache line replacement in L1, also between all repetitions performed. Otherwise, having a smaller data size that fits the L1 cache, all needed data, after being loaded a first time, will be already available for all subsequent repetitions, that will not cause cache misses any more. This is validated by measure results, that shows an `L1(miss)` counter value of 4 misses per loop, one for each `movaps` instruction. As we expected, no `L1(write-back)` are counted, because of the LOAD nature of the codelet, confirmed also by the `LOAD` counter, that shows 4 LOAD instructions per loop. Taking into account the total number of instructions present in each loop cycle, as showed by the instruction retired counter `IR` there are 7 assembly instructions in the innermost loop: 4 `movaps`, one for the counter *i* addition and two needed to implement the innermost while condition, namely a test instruction and a jump.

Another important result that table 5.3 shows is the difference between the energy consumed by the package, including the core and the cache energy consumption, and the RAM that is much smaller. This is an additional indicator on how much important is to optimize the CPU energy consumption, being the largest consumer of the system.

5.3 First results and interpretation

As explained in section 5.2 the objective of this step of our study is to understand how a core behaves with different data sizes and with different clock frequencies. This section is dedicated to the interpretation of obtained results.

5.3.1 Impact of frequency and data movement on energy

The first comparison that is possible to see is the graphic presented in figure 5.1. It gives an idea of the energy and execution time using different data sizes, that as we said act on the level of cache used. On the horizontal axis are present all different memory size used in the experiment. The left vertical

MEM(kB)	REP	FREQ(GHz)	RDTSC	URC	UCC	IR
86	370000	1,2	17,5437	17,4285	9,0931	7,035

LD	ST	L1(miss)	L1(wb)	L2(miss)	E_PKG(nJ)	E_RAM(nJ)
4	0	4,0117	0,0005	0,00	209,86	10,64

TABLE 5.3: Counters results for codelet

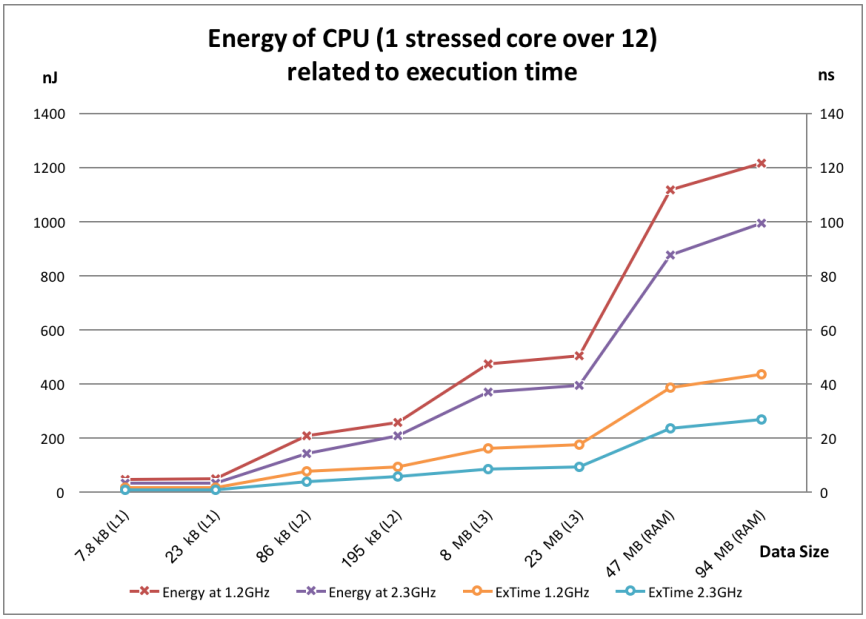


FIGURE 5.1: CPU energy (nJ) and execution time (ns) by level of cache

axis represent the energy consumption, expressed in nano-Joule, while the right vertical axis represent the execution time expressed in nano-seconds. The graphic shows four lines that, by couple, have different natures: the two on top refers to the energy consumption axis and the two at the bottom refers to the execution time axis. Putting our attention on the energy lines, the red one is the energy consumption when the core frequency is set at the minimal available frequency, in our case 1.2 GHz. The violet line, instead, is the energy consumed at 2.3 GHz, the highest frequency, with the TurboBoost deactivated. The same difference stands for the two line on execution time: the orange one is the execution time at minimal frequency and the light blue at maximal frequency.

A first intuitive result we expected confirmed by this graphic is that the more data the core have to deal with, the more time is needed. But more important, this relation is not linear. We can see, in fact, that the biggest changes in both the execution time and energy consumption are present when we change the level of cache used, while remaining in the same level of cache, even doubling (L2) or tripling (L1-L3) the data size used, there are almost no alterations in both energy and time; there are sensitive changes only when we work on RAM memory, anyway they are quite small compared to a data size almost doubled (from 47 to 97 megabyte). We can also see that the gap between caches grows with higher cache level, and is quite remarkable the gap between L3 cache and the RAM: this is due to the very different read/write speed of RAM modules in comparison to much faster cache memories.

This graphic, together with the one in figure 5.2, shows that, with this particular codelet, the choice to work at the lowest frequency doesn't save the overall energy consumption at all. In fact, it's clear that the orange line in the first graph indicates always higher values than the light blue one. This

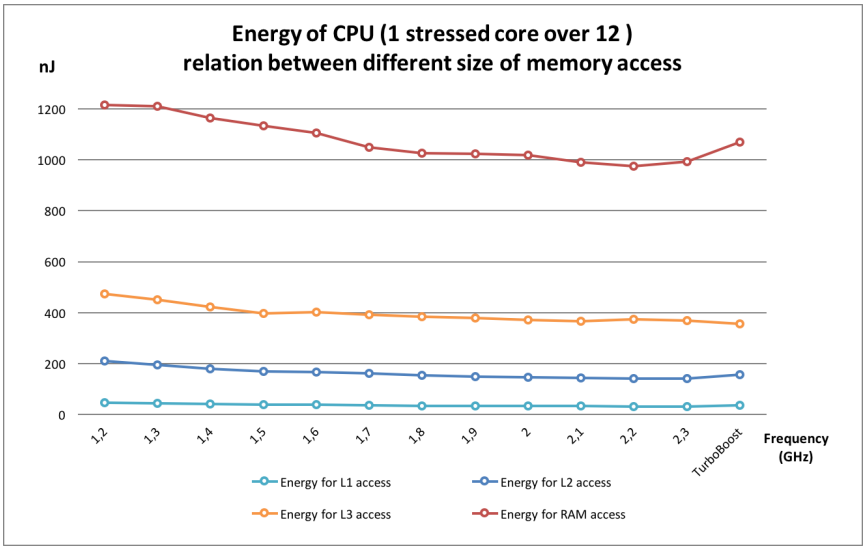


FIGURE 5.2: CPU energy by cache accessed and operating frequency

seems to be in contrast with equation 3.2, for which lower frequencies get lower $P_{dynamic}$. However, this is not the case, because as explained in section 3.4.1 we have to take into account also the time, using the general equation 3.1 for energy. In this example the higher power used to have an higher frequency on the core is less relevant than the time saved. Consequently we can stand that this codelet has a compute-bound behavior, especially due to the fact that is only one core active over twelve available, so that it can't saturate the memory bandwidth. It is very likely that running this codelet in parallel on all available cores, we will detect a memory-bound behavior.

Figure 5.2 also shows the consumed energy at every intermediate frequency, including TurboBoost. Different lines stands for codelet execution on different cache levels. This graph complete the information given by the last one, providing a vision of what happens by changing the frequency. As we can see, thanks to the speed on lower level of caches, executions on L1 and L2 are less sensible to frequency changes, even if we can anyway see a descending trend. From L3, and especially from the RAM, the energy consumption is lower on higher frequencies, with the exception of the TurboBoost, that we will detail later on. From this figure we can understand the importance of cache memories not only for performance, but also for energy savings purpose.

5.3.2 Energy, power and TurboBoost

Going further in the relation between energy, power and time, we analyze figure 5.3 and 5.4, in order better understand the relation expressed by equation 3.1 when we are in the most frequent use case in real parallel applications: frequent RAM memory access.

The first graphic shows the comparison between energy consumed and the execution time, while the second the relation with the latter and the

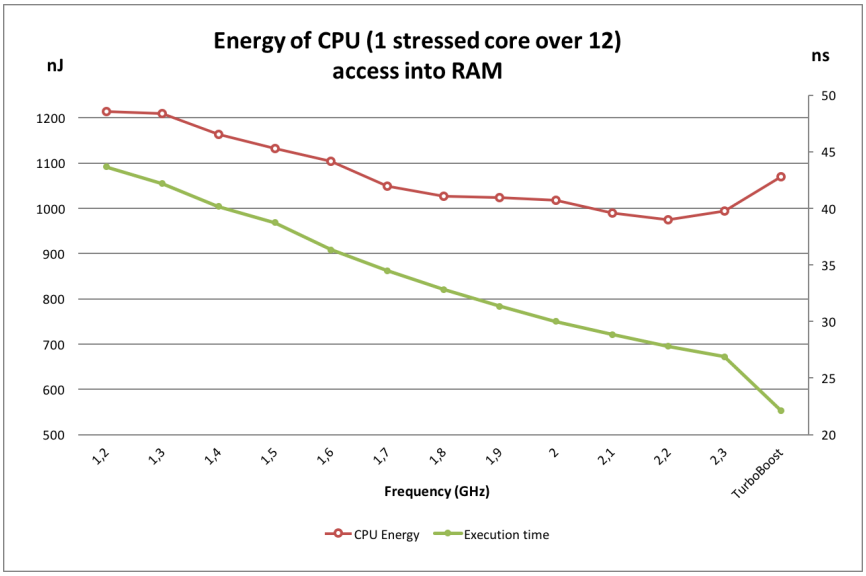


FIGURE 5.3: CPU energy and execution time by operating frequency

power, both of them for each operating frequency available on the node, represented on the horizontal axis. As before, the left vertical axis expresses the energy consumed in nano-Joule and the right horizontal the time in nano-seconds. In the first, the red line is then the series of energy value, in the second the blue line is the power and for both the green line is the execution time.

Those two graphic, together, not only confirm what we find out from previous results, but give us also an idea on how can be difficult to weight the frequency changes. In fact, from figure 5.3 we can see that in general the energy is lower for higher frequency used, because as we explained previously they lead to a faster execution. This consideration is applicable to almost all cases but TurboBoost: indeed the rightmost case in first graphic has higher energy consumption than 2.2 GHz and 2.3 GHz cases, even if it has a considerable drop in the execution time. The second graphic on power explains why. It express in fact the relation we presented in equation 3.2 between dynamic power consumption and operating frequency: the higher the frequency, the higher the power. As we can see from figure 5.4 this statement is valid for all frequencies, especially for the TurboBoost, where the increase of power consumption compared to other high frequency is much higher. This explain the previous graphic: according to formulas 3.1 $E = P \times T$, in this case the rise of power consumption is greater than the time gained time going faster. In fact, the TurboBoost is an Intel® technology that permit to overclock a core for a limited amount of time, depending on the type of workload, number of active cores, estimated power consumption and processor temperature. So power consumption of TurboBoost can vary a lot depending on the use case, but as we saw in our experiment (figures 5.3 and 5.4) it is not an energy-safe choice. It becomes reasonably inadvisable if the objective is energy saving.

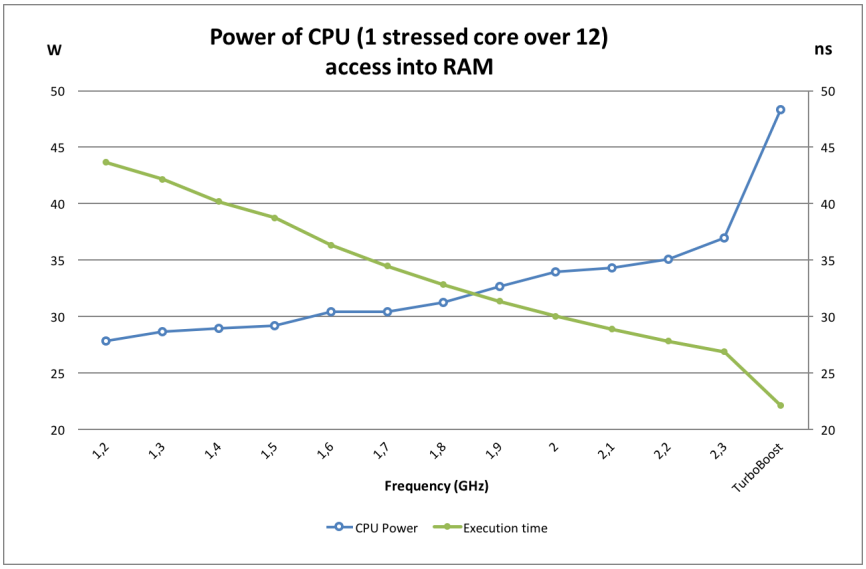


FIGURE 5.4: CPU power and execution time by operating frequency

With all the information provided by the study of micro-benchmark behavior through hardware counters, we have a good starting point to present the design of our tool: the Bull Dynamic Power Optimizer.

6 Bull Dynamic Power Optimizer

This chapter presents Bull Dynamic Power Optimizer, shortly BDPO, that is the experimental tool we developed during our internship in Bull. We will explain its company context, design and behavior, that is strongly based on all energy saving concepts we explained in previous chapters.

6.1 Product context

The idea for BDPO arrived in order to complete a suite of Bull products for energy savings. The final project is to create a unified framework for cluster energy management, that at the moment contains one product developed by the Power Efficiency team in the Bull Research and Development division: the Bull Energy Optimizer, called briefly BEO.

Bull have built supercomputers for decades, and with the coming of exascale computing, as we already explained, the need for hardware and software optimization for energy consumption is becoming more and more relevant and urgent.

BEO is an infrastructure-oriented tool, and it was started with four main approaches in mind:

- **Descriptive** approach: keep track of all possible energy consumption data and metrics on a cluster;
- **Diagnostic** approach: give limits a diagnosis on the whole system, knowing for example when it is in an energy-saving state or a energy-consuming state, or if the system is passing predefined thresholds;
- **Predictive** approach: understand with all collected information how the machines are consuming, being able to predict for given conditions the behavior of the cluster from a performance and energy consumption point of view;
- **Prescriptive** approach: Give hints to administrators and users to how they could reduce the energy consumption of their systems or jobs.

The idea behind BEO is then to offer to cluster administrators and users all the possible means to easily understand how and how much a big and complex cluster consumes, at a physical and logical levels: the customer can figure out which parts use more power, either a switch, a computer node or a storage node; it can also estimates the energy cost in terms of money for a specific job or for a part of the cluster for a certain period. Bull Energy optimizer is nowadays ad the diagnostic level, continuously evolving to reach the predictive and prescriptive features.

In this framework, the Bull Dynamic Power Optimizer was started with this internship as application-oriented tool next to BEO, in order to explore the potentiality of direct energy optimization of parallel applications, starting from the DVFS technique.

6.2 BDPO design

In order to obtain the maximum performance with the minimum performance degradation and resource demanding on a node of a cluster, we decided to design this utility with the C programming language, that permits to achieve high performance and write optimized code for very lightweight memory usage.

Even if the C programming language doesn't support the Object Oriented Programming paradigm, that easily leads to a modular design, we choose to develop this project in a modular way, designing simple and effective interactions between modules, paying particular attention to those module that have to run efficiently at runtime.

We decided, moreover, to design a tool that will be distributed and installed on each node of the cluster, so that each instance will be responsible of the monitoring of its own node, avoiding message passing between nodes and a centralized solution, that could lead to bottleneck problems and performances issue. This approach is not too far from the concept of autonomic computing [26]. An autonomic system can be modeled in terms of an independent control loops with sensors (for self-monitoring), effectors (for self-adjustment), knowledge and planner/adapter for exploiting policies based on self- and environment awareness. This architecture is sometimes referred to as Monitor-Analyze-Plan-Execute (MAPE). It is somehow possible to associate our modules to sensors, effectors, planner and adapter, but we did not design BDPO modules to fit the MAPE model.

The diagram in figure 6.1 shows the software solution we designed. The following is a brief description of each module:

The main

This is the launching program: it initializes all other modules, launches the Control Loop and takes care of the finalization of modules.

Control Loop

The Control Loop is the core of this design, the module that act as fulcrum between all other modules. It is in charge of looping at a specific polling period, calling functions and asking values to all other modules, depending on parameters received by the configuration module, returning values from other module and process logic that we will describe in section 6.3.

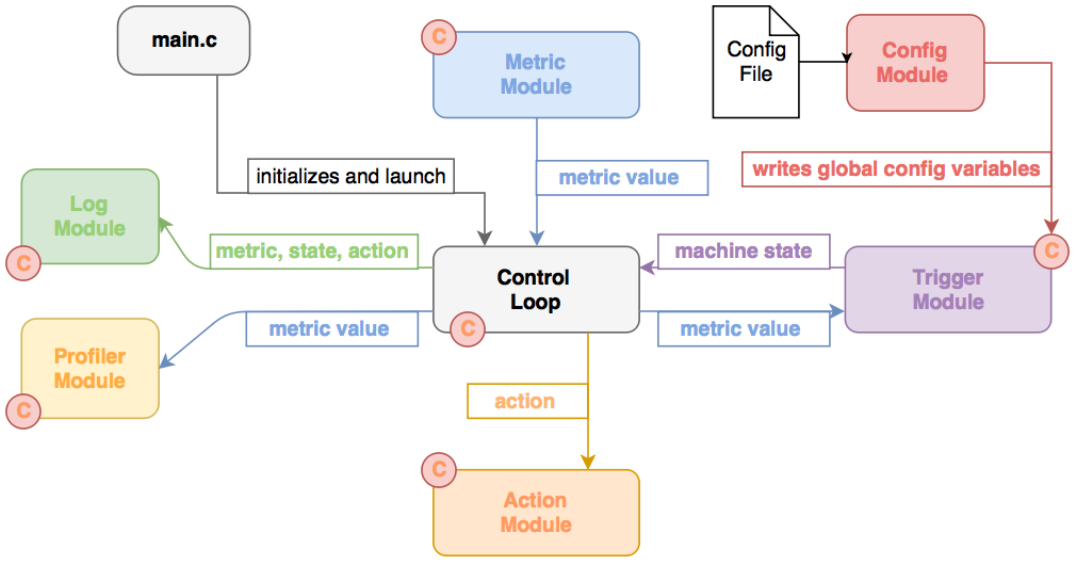


FIGURE 6.1: BDPO modular design with relations between modules

Metric Module

This module contains the logic behind metric extraction. It manages already presented counters in order to obtain metrics to be delivered to the Control Loop. A detailed description of this module is provided in section 6.4.

Trigger Module

The Trigger is the module that will receive from the Control Loop the monitored metric and figures out the state in which the machine is going in or it is coming from. For now it is implemented in the simplest way possible: it receives a threshold from the configuration file and simply answer to the Control Loop if the received metric value is above or below the predefined threshold. It keeps a history of the previous state, so that it can tell if the machine is passing from the "lower state" below the threshold to the "upper state" above it, and *vice versa*.

Action Module

This module have to define the actions that have to be performed on the machine depending on the state detected by the Trigger. We give more details on actions in section 6.5.

Log module

If activated in the configuration file or the command line, the logger is in charge of keeping track of the history of all actions performed by BDPO, depending on the machine state and the metric value. The produced log file could be useful to better understand and analyze the behavior of BDPO on

different applications offline. It uses an optimized buffer in order to avoid high number of system calls at runtime.

Profiler module

As the Logger, the Profiler can be activated by the configuration file or the command line options. It is the main tool we used for all experiments we will describe in chapter 7, as it permits to keep track of all metric value retrieved by the Metric module. It also used an optimized buffer, and we will better describe this module in 6.6.

Configuration Module

This is the module that is in charge of parsing the configuration file and the command line options, in order to create a global structures of parameters that can be used by all the other modules in the system. Options include names of output files, the polling period for the Control Loop, the threshold used by the Trigger and so on.

6.3 Control loop

The control loop is the main manager module. It is responsible to coordinate all the other modules, which communicate through it. It has also to activate the right module depending on the parameters passed from the configuration file or the command line. Listings 6.1 shows the Control Loop pseudo code:

```
1  int bdpocontrol() {
2
3      metric = bdpoconfig_get(metric);
4
5      bdpometric_load(metric);
6
7      bdpometric_start(metric); // Start metric monitoring
8
9      do { // main Control Loop
10         bdpometric_get(metric, &metric_value);
11
12         if (is_action_active) {
13             state = bdpotrigger_state(metric, metric_value);
14
15             switch (state) {
16                 case STATE_CROSSED_UP:
17                     return_code = bdpoaction_freq_up();
18                     if (is_log_active) {
19                         return_code = bdpolog_log(&log_data);
20                     }
21                     break;
22
23                 case STATE_CROSSED_DOWN:
24                     return_code = bdpoaction_freq_down();
25                     if (is_log_active) {
```

```
26                          return_code = bdpolog_log(&log_data);
27                      }
28                      break;
29                  }
30              }
31
32              if (is_profile_active) {
33                  return_code = bdpoprofile_profile(&profile_data);
34              }
35
36              sleep(sleep_period);
37
38          } while (!stop_main_loop);
39
40          bdpometric_stop(metric); // Stop metric monitoring
41
42          return (return_code);
43      }
```

LISTING 6.1: Control Loop pseudo code

As we can see from the pseudo code, the first two functions take the desired metric from the configuration module and than load it into the metric module. Then, the monitoring of the specified metric is started.

The core of this module is obviously the central loop. As the listings shows there are a lot of conditions: those are all previously taken from the configuration file or the command line through the configuration module. Those condition are necessary to run every module only when necessary, and to manage the logical dependencies between modules[1]. The main loop gets a metric value and then pass it to the Trigger. The Trigger will then evaluate the current value of the metric with the past node states, returning a new state. The latter is then used by the Control Loop to understand which action is the right one to call. For now only the two action related to target frequencies for DVFS are implemented, as we will detail in 6.5. If needed, the system will also log the decisions and actions taken. Last, the loop will call the profiler if the related option is true.

When the main loop is stopped by a signal from the user, it simply stops the monitoring of the metric and then returns.

6.4 Metric extraction

The main idea behind the implementation of the metric module is to create an infrastructure that is by itself expandable. In our case, for instance, we created an interface that can be called for different type of metrics, completely independent to each other. As we saw from listings 6.1 at line 5, there is a call to `bdpometric_load(metric)` that permits through pointers to functions

[1]For instance, if the action module is deactivated, there are no reason to activate the trigger module because the trigger decisions are necessary only for deciding which action is needed to execute

to call all the other functions related to the metric to the desired implementation. This technique allows to have completely independent implementations of different type of metrics, with the use of different technologies, counters or interfaces. With the function call described before we simply select the desired metric to be monitored.

6.4.1 Instruction per cycle

The first metric we implemented for BDPO is the Instruction Per Cycle (IPC) metric. The idea behind the use of this metric is to being able to have a quantity that gives us a measure of the actual workload on the nodes. The objective of this metric is then to help to discriminate between compute phases and not-compute phases.

As we explained in section 3.4.1, for compute-bound phases higher frequencies are the best choice for energy consumption, because they reduce the execution time. Instead, for memory-bound phases or non-compute phases in general we know that lower frequencies permits energy savings, while they use lower power consumption during the phase. We used this logic to call the right action to be performed on the cores, as already explained in previous Control Loop description.

The IPC is computed using some of the counters described in 5.1.1, as follows:

$$IPC_{node} = \frac{\sum_{core}^{tot\_cores} \texttt{INSTRUCTION\_RETIRED}_{core}}{\texttt{TSC}}. \tag{6.1}$$

We recall that nowadays compute nodes, especially in HPC systems, have multi core CPUs. Moreover, most of the counter previously described are available for each core, and this is the case for the `INSTRUCTION_RETIRED` and `RDTSC` counters. For those reasons, we had the choice between defining a metric with a per core granularity or an aggregated metric per node. For sake of lightness and runtime performances of our tool, we made the decision to define an IPC per node. This choice define also the granularity that BDPO have to use in order to take an action on the hardware, as described in next section.

As we can see from equation 6.1, then, to compute the node IPC we sum the value of the `INSTRUCTION_RETIRED` counter over all the available cores, dividing it for the total number of cycle occurred during the desired period of time.

Figure 6.2 shows an example of what the IPC monitoring gives for a matrix multiplication application implemented with MPI and the `dgemm` routine [27]. The diagram shows the IPC detected on a single node with 20 cores working on the matrix multiplication. This example clearly shows a sequence of different compute phases: it alternates `dgemm` multiplications with smaller matrices and bigger matrices. Working on smaller matrices causes an higher IPC, because small data size takes advantage from low-latency lower-level caches. Instead, when the program has to compute bigger matrices that not fit the caches, it has to work more into the RAM memory. RAM has

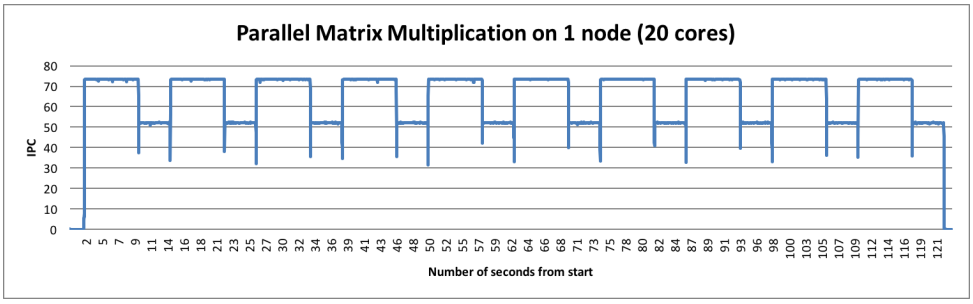


FIGURE 6.2: IPC profile of a simple MPI application

higher latency that slows down the core execution pipeline, causing a lower IPC profile.

This is a clean and simple example, and we will present in chapter 7 what this metric gives as result during the execution of real HPC applications.

6.4.2 Memory activity

We also started studying another metric, related to the memory stress. In particular, the idea was to understand the data flow activity between core-related caches and the uncore-related caches. This data traffic is usually between the L2 and L3 caches, because the L2 cache is private to the core and the L3 cache is shared between all cores in the same package.

The memory activity metric takes into account both write and read operations the L2 cache performs on L3 cache. This metric is computed by

$$L3\_ACTIVITY_{node} = \frac{\sum_{core}^{tot\_cores}(\texttt{L2\_WB}_{core} + \texttt{L2\_RQST}_{core})}{\texttt{TSC}}.$$

In our studies we haven't reach a good level of testing of this metric in order to implement optimization strategies related to memory metrics, so that all the studies presented in the last part of this thesis will refer only on the already presented IPC metric.

6.5 Taking an action

As we already said, the action module is in charge of the definition of all possible actions that can be performed on a compute node, so that using those actions in the correct application phase, power saving is possible. Like the metric module, we designed the action module with modularity in mind, so that it is always possible to enlarge the set of actions to be taken.

As we have largely explained in chapter 3, the DVFS is the target mean for energy optimization for this project. We had then to implement a method that permits to scale the core frequency to a desired frequencies.

In section 4.2 we have already presented the *CPU frequency scaling* driver for the Linux environment, where we have the possibility to choose in between multiple frequency governors. Setting the `userspace` frequency governor, it is then possible to choose the operating frequency of each core either by calling a driver function or by directly writing specific virtual files with the desired frequency. Those virtual files, one per core, are used by the `acpi-cpufreq` to set the operating frequencies, only when the `userspace` governor is active. We recall that dynamically scale the frequency is not a cost-less process. There is an overhead due to frequency transition, as described previously in 3.5.

From listing 6.1 we already saw the two pseudo functions related to the `bdpoaction` module: `freq_up()` and `freq_down()`. The strategy behind this design is that during the initialization phase of BDPO, the action module takes from the configuration manager the high and low frequencies to be used by those two functions. We will details further the process of choosing of high and low frequencies in chapter 7.

As we explained, at the current version the BDPO use the DVFS techniques with a fixed high frequency and low frequency, but there exist other techniques and actions to be explored in the future, notably the scaling of uncore operating frequency and the clock modulation [28].

6.6 Integrated profiler

The profiler is a tool that permits to follow the trend and behavior of metrics on-the-fly during a program run. Figure 6.2 is an example of profiler use: after the execution of the matrix multiplication program, we can see how the IPC changed during the run, allowing deeper application understanding.

We decided then to integrate this feature to BDPO, adding the possibility to activate the dump of the retrieved metrics values in an output file. This is showed also in lines 32-33 of listing 6.1, that contains the conditional profiler call.

A really important point of a profiler is that it must be as lightweight as possible, so that perturbation of application execution is minimized. We also know that writing values into a file involves a system call, that is quite heavy in terms of performances. Moreover, it is possible to write values either in text format or binary format. Translation from numeric values to chain of characters is also a time consuming operation, especially when done quite often.

We designed then a lightweight profiler that relies on an optimized fixed size, binary buffer.

The optimized buffer

This solution permits to reduce a lot the number of system calls to perform input/output operation of files, and totally avoid the overhead introduced by the string conversions. We can see from Control Loop pseudo code that at each loop, a metric value is requested to the metric module. Then, if the

purpose is to only profile the application, no action are needed, so that all the logic behind the combination trigger-action is skipped. The metric value is then passed at the profiler in its binary form, in the code represented by the structure `profiler_data`. No translation is needed and, furthermore, the size of the structure is fixed. The profiler internally, put this value in the buffer, which size is always known and surely a multiple of the data size, thanks to the smart initialization of the buffer. This technique prevents the buffer to check against overflows at each add of a new element. Once the buffer is full, it performs a system call to write its whole content on the output file.

The translator

During the whole execution of the application together with BDPO, only binary values are stored on disks. So how to read it? We designed an internal binary-to-string translator, that it will be called either at the end of the overall execution (when the target application has already finished its work, so no high performances are needed anymore) or on demand, to translate already produced binary files. This design allows then high performances and lightness at runtime and human-exploitable results when needed.

7 Validation on real HPC applications

In this chapter we will briefly present some real HPC applications we choose for the validation of the BDPO tool, explaining the objective we had on those targets. We then present the results we obtained for each application, followed by an analysis of the results.

7.1 Performance metrics

HPC applications are, in general, quite complex by themselves: they are domain specific, with usually an elaborated software structure, with algorithms designed to be as parallelizable as possible. From and external point of view, it is often quite hard to understand in deep what they do, especially trying to know how they behave at runtime. For example, one can expect that a parallel application running on multiple homogeneous nodes it acts homogeneously on all of them. This is not always the case, as workload unbalance may happen.

The third objective we presented in chapter 2 for the BDPO project helps to understand the parameters and metrics we used to analyze the results we will presents in this chapter. In that point, we said that we wanted to reach almost no performance degradation for the target application. As we said many times, in the High Performance Computing environment, as the name suggests, performances are crucial. With our tool we wanted to save energy on a real HPC system running a real HPC application, but an important point was to do not affect the system performances, or at least as less as possible and the closest to zero.

For parallel software, in general, performances are evaluated as the execution time. The shorter the execution time, the higher the performance. Then, we used the Slurm workload manager, as described in section 4.1.2, to measure the energy consumed by each job.

Asking applications experts inside the company, we figured out that in the HPC environment, roughly speaking, the 80% of the time systems are configured to run with the `performance` frequency governor, that set the operating frequency always at the highest frequency available. For the remaining 20% of use cases, the `ondemand` governor is used, that tries to perform some frequency scaling based on the workload. So we took those two cases as reference, and we compared the application run with BDPO active with both independent run using the two mentioned frequency governors.

Every application is different from the others, so we didn't expected to be able to reach energy saving on all of them. For this reason, we have set the constrains to be, at least, conservative on a program execution. This means that, comparing a run with BDPO with an execution with either the `performance` or the `ondemand` governor, the performance overhead introduced by BDPO it is in an acceptable range.

7.2 HPC applications

Here is a brief presentation of the HPC application we used for our tests.

GROMACS

The GROningen MAchine for Chemical Simulations (GROMACS) [29] is a molecular dynamics package mainly designed for simulations of proteins, lipids and nucleic acids. It means that it is able to simulate the Newtonian equations of motion for systems with hundreds to millions of particles. It was originally developed in the Biophysical Chemistry department of University of Groningen, and is now maintained by contributors in universities and research centers worldwide. GROMACS is one of the fastest and most popular software packages available, and can run on both CPU-based and GPU-based clusters. It is free, open-source software released under the GNU Lesser General Public License.

It is primarily designed for biochemical molecules that have a lot of complicated bonded interactions, but since GROMACS is known to be fast at calculating the non bonded interactions (that usually dominate simulations) many groups are also using it for research on non-biological systems, for example polymers.

WRF

The Weather Research and Forecasting (WRF) Model [30] is a next-generation mesoscale numerical weather prediction system designed for both atmospheric research and operational forecasting needs. The model serves a wide range of meteorological applications across scales from tens of meters to thousands of kilometers. The WRF developing process began in the latter part of the 1990's and was a collaborative partnership between a lot of different entities, like the National Center for Atmospheric Research, the National Oceanic and Atmospheric Administration, the Air Force Weather Agency, the Naval Research Laboratory, the University of Oklahoma and the Federal Aviation Administration.

For researchers, WRF can produce simulations based on actual atmospheric conditions (observations and analyses) or idealized conditions. WRF offers operational forecasting a flexible and computationally-efficient platform, while reflecting recent advances in physics, numerics, and data assimilation contributed by developers from the expansive research community.

NEMO

The Nucleus for European Modelling of the Ocean alias NEMO [31] is a modelling framework of ocean related engines, aiming at modeling a set of oceanic events and phenomena for research purposes. It is developed and maintained by the NEMO European Consortium, involving France, Italy and the United Kingdom (CNRS, CMCC, Nerc ...). It has been developed with a modular approach, and contains several different modules with different purpose, notably: GYRE is dedicated to simulate oceanic whirlpool; OPA is dedicated to simulate ocean dynamics (called "the blue ocean component"); LIM simulate the phases equilibrium between ice and water around the poles (called "the white ocean component"); TOP is dedicated to simulate the geological and chemical mechanism which have an impact on the oceans, such as CO2 absorption (called "the green ocean component"). Those modules may be executed separately or together.

The range of applications includes oceanographic research, operational oceanography, seasonal forecast and (paleo)climate studies. Used by a large community of users since 2008, a lot of projects have been carried out and about 300 publications have been published using the framework.

7.3 Results presentation and analysis

In order to test the BDPO tool on the presented applications, we had to understand how those applications work. First, we had to compile the source code, with the necessary libraries and compilation tool set. Second, we had to understand how to run the application in its simplest form. Third, a launching parameter study was necessary in order to find a reasonable and plausible run configuration, in term of nodes, number of processes, and computing parameters. Fourth, a profiling of the program with different operating frequencies was needed in order to know how the application itself behaves at runtime, and to understand if it had clear memory-bound or compute-bound functioning. In those cases, we already know that setting the system to respectively the minimal and the maximal operating frequency ensure an optimal energy consumption. Fifth, if we found that the application has a balanced behavior, the next step was to study it with almost all available operating frequencies, comparing the energy consumption of each run in order to find the best frequency to be used by BDPO. Finally, we performed a set of runs with both `performace` and `ondemand` governor for reference and then with BDPO active to retrieves our results. The following are results presented by application.

7.3.1 GROMACS

The set up of GROMACS was done by means of a Bull internal framework, mainly created for taking measures on real applications. This framework was useful for us in order to avoid a lot of compiling and running issue. This execution framework fixes the execution time of a run at around 5 minutes.

Frequency	1.2 GHz	2.3 GHz
ns/day	0.968	1.776

TABLE 7.1: Performance results for GROMACS application

Then, depending of the available resources, like nodes and cores on each node, the application performs a different workload, which fits the requested execution time. Using this technique, the more are the available resources, the bigger is the workload.

Having a fixed execution time prevent us to provide a comparison between execution time of different runs. However, to solve this problem, the application gives an output parameter which expresses the overall performance of the application for a run: ns/day. This parameter specify the number of nanoseconds of chemical reaction that the system will be able to compute for a 24 hours of calculus.

We use then this parameter as index for performances. The higher the ns/day the higher the run performance.

Experimental Results

The first test we performed on GROMACS, as explained above, was to understand if this application already had a well-known behavior. To check this, we used the framework described before to launch GROMACS with two different operating frequencies: the minimal and the maximal one, respectively 1.2 GHz and 2.3GHz. The framework automatically balanced the workload based also on the chosen operating frequency in order to give two run with a duration of 5 minutes. Tests were performed on a compute node with 20 physical cores, with only one process pinned per core (we did not used Hyper-Threading), and TurboBoost deactivated. Results are presented in table 7.1.

These results express the CPU-bound nature of the GROMACS application: we can say that with almost doubled frequency, we obtain almost double performance. This means that the "bottleneck" in the application is almost only the computation phase, so that the quicker we compute, the quicker we complete the execution. In this type of application, accordingly with theory explained in chapter 3, we know that the maximal frequency is always the best choice in term of energy consumption and performance.

IPC profile

We also analyzed the IPC profile of GROMACS: figure 7.1 shows the overall evolution of the IPC during an execution at maximal frequency. The horizontal axes represents the time, with the number of seconds passed from the profiler start. On the vertical axes there is the IPC values. As we can see, the IPC has the same trend during the entire duration of the application, continuously oscillating between 30 to 40 IPC.

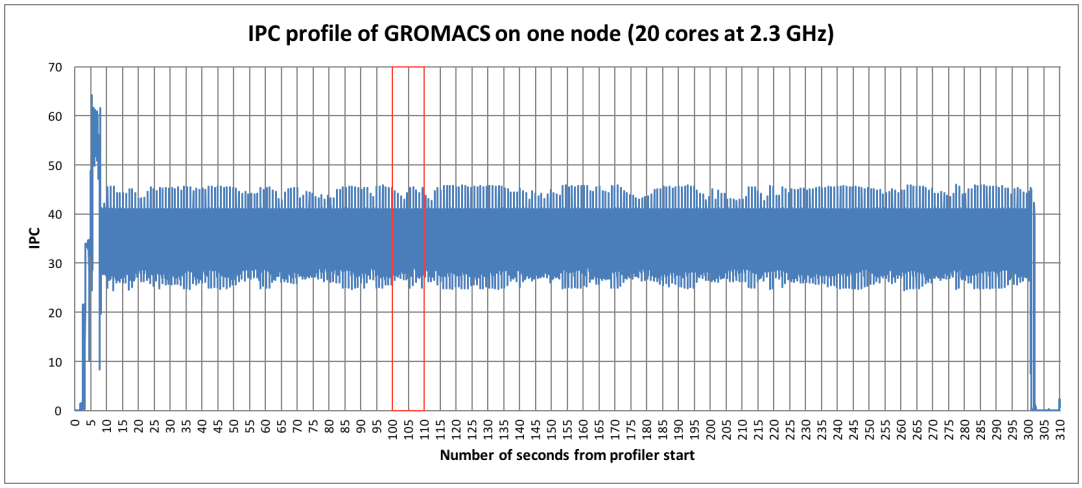


FIGURE 7.1: GROMACS IPC profile

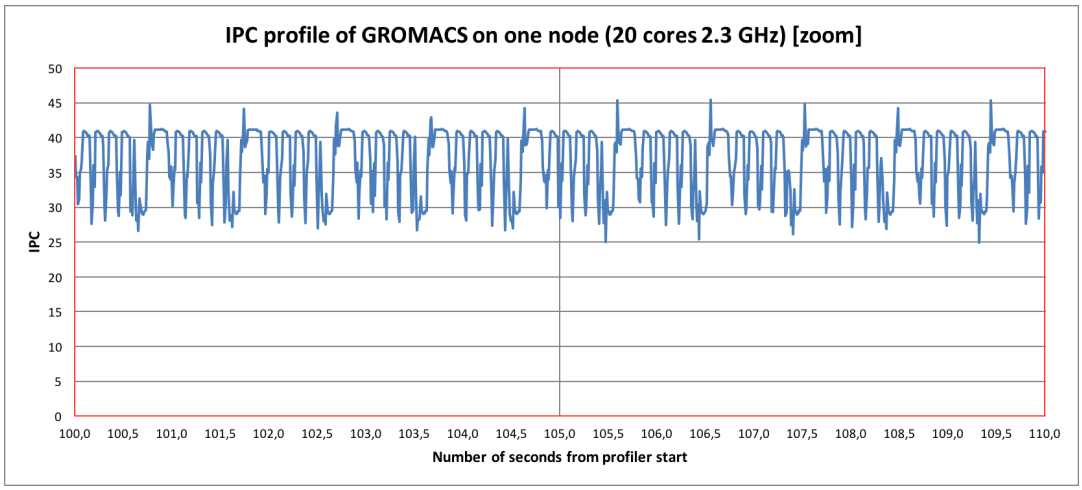


FIGURE 7.2: GROMACS IPC profile (zoom)

If we want to have a closer look, figure 7.2 shows with a different horizontal scale what happens during the 10 seconds highlighted with a red box in figure 7.1.

First, the second image shows a clear periodicity into the profile. This is probably due to the characteristic iterative approach of parallel applications. Moreover, we can also see that the IPC almost never get down below a value around 28-30 IPC, suggesting a permanent intensive-work phase. The relatively small oscillations in the profile are very likely do to message passing between processes, but as we have seen from performance evaluation before, they are not enough substantial to cause a memory-bound behavior.

Because of the presented study, understood that the GROMACS application is CPU-bound and the frequency that best fit in this case is the highest one, we didn't apply the BDPO tool on the GROMACS application.

7.3.2 WRF

The WRF application has many compilation options and possible usage. We compiled WRF for Idealized Data Cases and distributed memory option (using the MPI library), and we performed our test on the 3D baroclinic waves case[1].

We then performed some preliminary tests in order to tune the parameters. The objective was to have an overall execution time of 10 minutes. We found that using two homogeneous nodes of our cluster, both of them with 20 physical cores, and setting the WRF internal domain parameters `e_we`, `e_sb` and `e_vert` with a value of 200, we had an execution time of around 10 minutes.

Frequencies comparison

After first tests, we execute a set of WRF runs in order to explore the behavior and performances of the application with all available frequencies, as described before, replicating the set 3 times for reproducibility purpose. In all those experiment our tool was deactivated.

Figure 7.3 shows the results for the runs with all available frequencies. The chart in the figure presents, for all frequencies marked on the horizontal axis, the energy consumption and the execution time of the job. On the left vertical axis there are the energy values in kilo-Joule, represented with the green line on the chart, while on the right vertical axis are presented the execution time values in seconds, represented with the blue line in the chart. The values on the chart represent the median of the 3 set of run repetitions.

We can see that the best performances in terms of execution time are reached with the highest frequencies: 2.2 GHz and 2.3 GHz. In fact, they have almost the same value. On the other hand, best performances in terms of energy consumption are reached using one of the middle frequencies, more precisely 1.8 GHz. This result tell us that WRF, in the studied case, is a balanced application, that is to say that it isn't a CPU-bound application neither a memory-bound application: even if the blue line (the execution time) suggest a descending trend while raising the operating frequencies, typical of CPU-bound behavior, the energy line give us other important information. In fact, it tell us that going from lower to middle frequencies (left half of the chart), the application behaves like a CPU-bound one, because also the energy consumption goes down with the execution time. Things change in the right half of the chart, where the reduction of the execution time due to higher frequency is no more sufficient to have an energy gain. It is very likely that from 1.8 GHz to the maximal frequency we start to saturate the memory bandwidth, causing a memory-bound behavior.

Those results were crucial to select the two frequencies needed to launch BDPO. As max frequency, useful for compute phases, we selected 2.2 GHz because it is the frequency for which we found the best execution time and energy consumption better then the maximal one. As min frequency, to be

[1]For reference, the cited case is related to the code `em_b_wave` inside the WRF framework

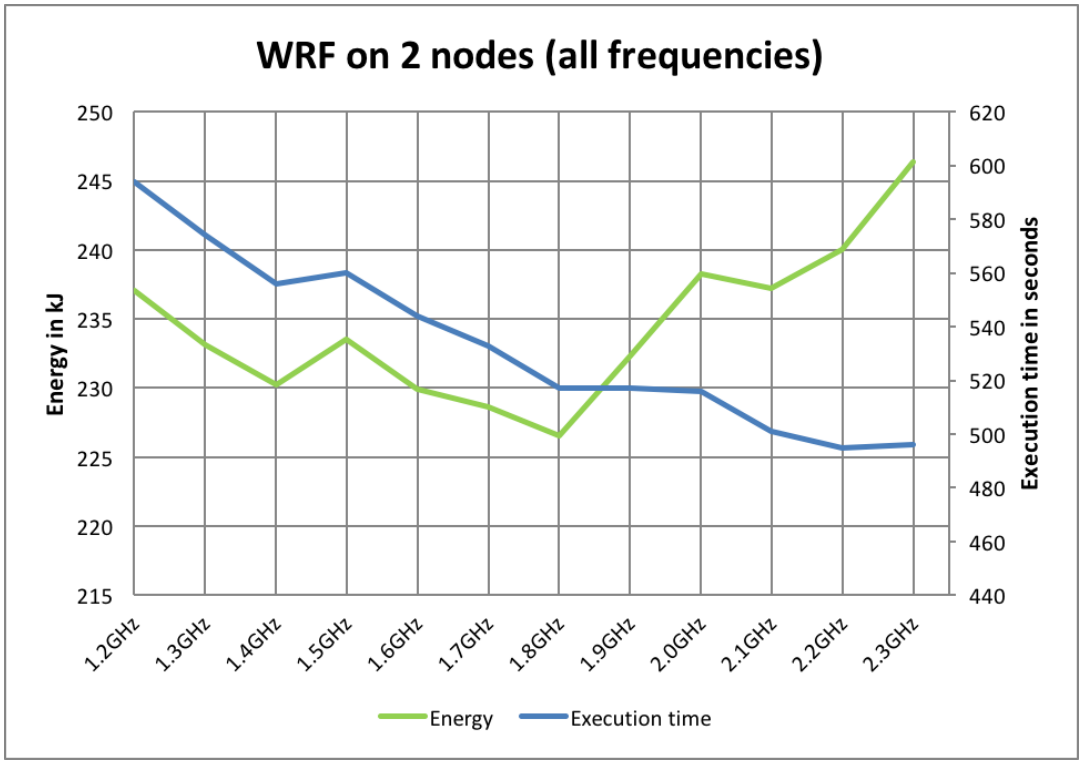


FIGURE 7.3: WRF frequency comparison

used during memory or message-passing phases, we choose 1.8 GHz because it is the frequency with the lowest energy consumption and with a reasonable performance degradation for the execution time.

IPC profiling

We then need to study the IPC profile of WRF in order to find out the threshold to be used by BDPO. We recall that the threshold is necessary for BDPO to discern between compute phase and memory phase. With this methodology, the tool will apply on the node the lower frequency for the memory phase and the higher one for the compute phase.

The charts in figures 7.4 and 7.5 shows respectively the IPC profile of an execution of WRF and a scaled version, a sort of zoom on ten seconds in order to better understand the IPC trend: the run is in parallel on two computation node, each one of them with 20 cores all set at maximum frequency. The two IPC profiles[2] are in this case overlapped, with the purpose of coherence-checking between nodes. As always, the horizontal axis represents the time in seconds while in the vertical axis there are the IPC values. The red and blue lines, almost overlapped, are the IPC profiles for the two different nodes.

The general view shows three little phases, at the beginning, in the middle and at the end, where the IPC is really stable, around a value of 35 IPC.

[2]We recall that the BDPO profile run on each node, so that during an execution in parallel on two machines we have two different IPC profile for one execution.

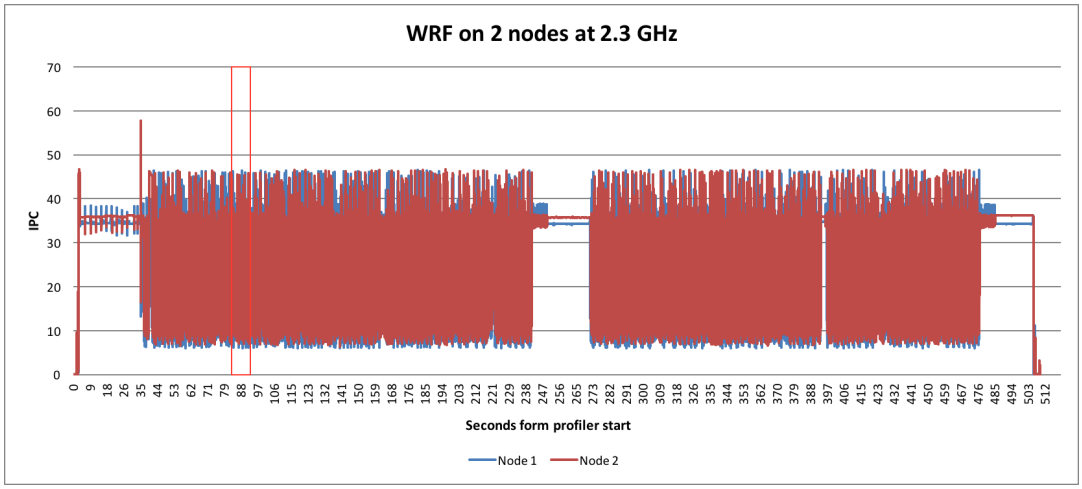


FIGURE 7.4: WRF IPC profile

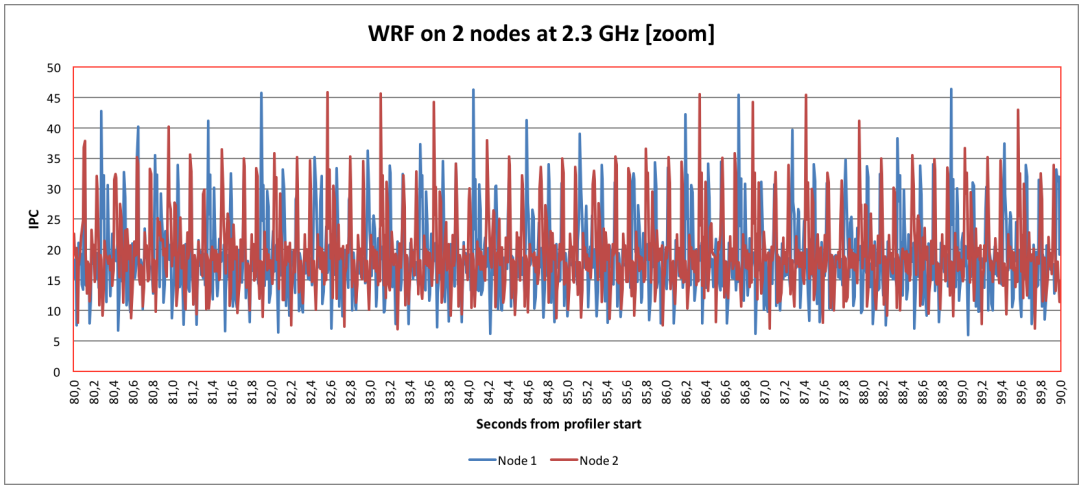


FIGURE 7.5: WRF IPC profile (zoom)

Those probably correspond to some synchronization phases between processes. Another possibility is that they are write-on-disk phases, but we exclude that because in other test, that we will describe better in section 7.3.3, we saw that during writing phases the processors go to C-states, and the IPC drop down to zero. Other than that, two big computation phases are present, where we can clearly see an oscillation of IPC, actually larger than the GROMACS case study, from below 50 IPC to below 10.

The zoom view shows a slice of 10 seconds, highlighted by the red box in the first image, during the computation phase from second 80 to second 90. As we can see, the profile presents a lot of drops and raises with a very variable IPC, that swing in a range around 10-40 IPC. The IPC profile is much less regular than the one seen with GROMACS.

The previous described profile was obtained by taking one sample each 10 milliseconds. We than though to augment the sampling frequency in order to see a better profile. Figure 7.6 shows a profile comparison between the one already described and another one captured with the same condition

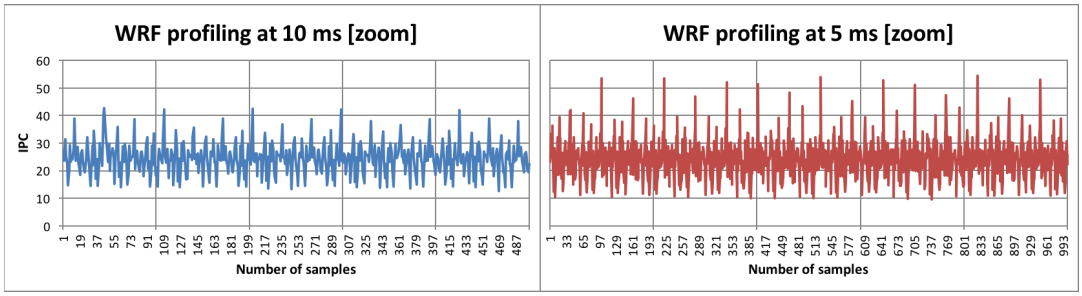


FIGURE 7.6: WRF sampling frequency comparison

Job ID	Execution time (s)	Energy consumption (kJ)
3180	516	245.1
3181	517	246.5
3182	525	249.8
3183	518	246.7
3184	519	247.5
median	**518**	**246.7**

TABLE 7.2: WRF with BDPO set of run

as before but the sampling frequency, that is set to take a sample each 5 milliseconds instead of 10. The comparison tell us that apparently the WRF has a really fast variability, that did not permit to clearly see some phase alternation.

BDPO Results on WRF

We tested WRF with BDPO activated, with a threshold at 20 IPC. We recall that we choose as low frequency 1.8 GHz and as high frequency 2.2 GHz. For reproducibility, we performed 5 runs with BDPO active, where the obtained results are presented in table 7.2.

We compared results with BDPO to results taken at the most used case in nowadays clusters, with the `performance` frequency governor, that set the operating frequency at the maximal available. As we can see from table 7.3, the BDPO tool is conservative for the energy consumption. On the other hand, for the studied case it suffers a performance degradation of around 5%. This is probably due to the high variability of the application IPC, that during the run triggers too many times an action performed by BDPO. In this particular case our tool simply continuously try to change the operating frequency too often, not permitting to gain energy from the change. This also probably cause the 5% of overhead. We already described this type of issue in section 3.5.

7.3.3 NEMO

NEMO was an interesting application to study, because it gives to the user some features that aren't available for other applications. First among all,

Set	Execution time (s)	Energy consumption (kJ)
`performace`	495.3	246.2
BDPO (median)	518	246.7
Comparison	+4,58%	+0,19%

TABLE 7.3: WRF results compared to reference

with NEMO we have the choice to set and choose how many intermediate checkpoints are performed during the computation by the application.

A checkpoint is a moment during the execution of the application where the program itself dump on disks all the results founded so far. This can be really helpful in the case a job lasts for hours, days in some cases. If a failure of the system occurs, all the work already done is lost. Instead, having some periodical checkpoint, after a failure the program can restart from the last checkpoint rather than restarting from scratch.

For the NEMO application we had then the possibility to tune different run parameters, especially three: the domain size, the number of intermediate checkpoints and the total number of iterations. The iterations specify the subsequent time slice of simulation to be processed: more the iterations, more the time simulated. We found a reasonable combination of parameters, to be run on 6 homogeneous computing nodes: with a domain size of 60, we set 2 intermediate checkpoints for a total of 300 iterations. The six mentioned nodes were all homogeneous, with 24 physical cores on each node.

Frequencies comparison

As we did for all studies applications, we ran NEMO with the described configuration, on six nodes, using all available frequencies. Figure 7.7 present a chart with results obtained running NEMO on each operating frequency.

As for the WRF case, this chart is organized with all frequencies on the horizontal axis, the energy values on the left vertical axis and the execution time values on the right vertical axis. As before, the energy line is colored in green while the execution time is the blue line. Unfortunately, we have one probable outlier at 1.3 GHz, that seems to mark a considerable drop in the consumed energy for a job duration that is way higher than all the others. We added then a dotted line in order to highlight the most probable trend of the two lines.

This chart help us to understand the NEMO functioning. As we can see, going from lower to higher frequencies, both the lines seems to have a linear behavior, even if the execution time is always decreasing while the consumed energy is almost always increasing. For a compute-bound application, as explained in chapter 3, we would expect a lower energy consumption for higher frequency because the job is completed faster. This is not the case for NEMO. By the way, for a memory-bound application we would expect an execution time almost invariant on different used frequency. Also this is not the case, even if is closer than the compute-bound case. We reach then

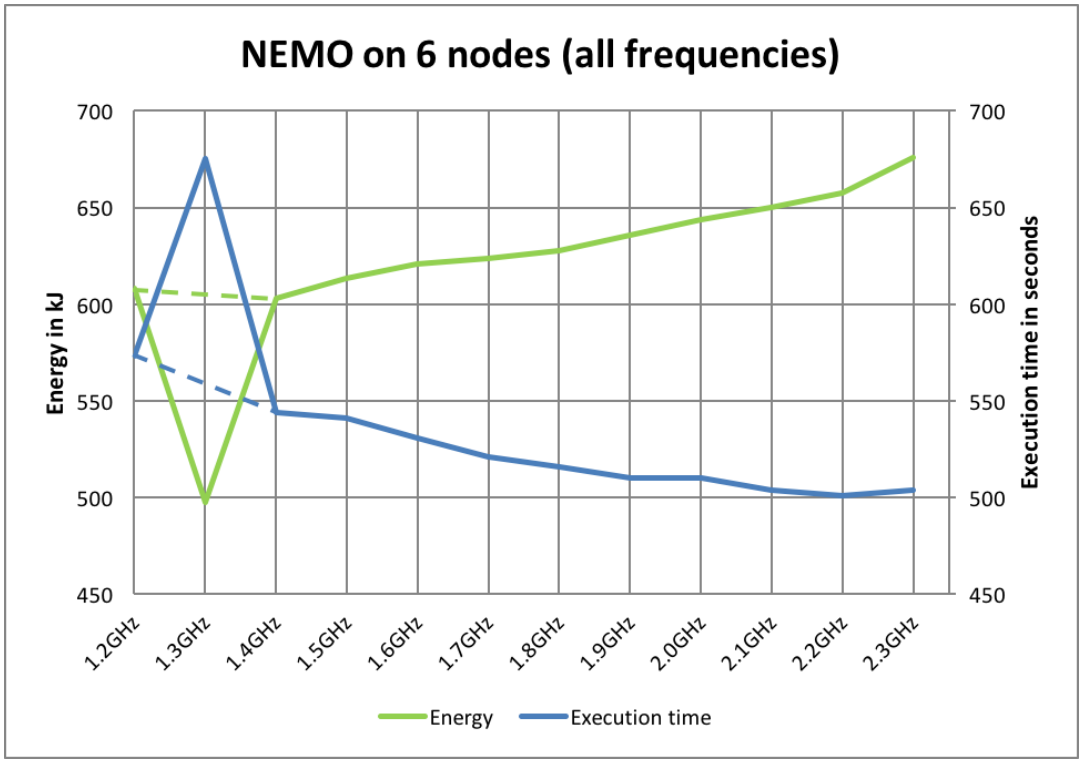


FIGURE 7.7: NEMO frequency comparison

Frequency	Execution time (s)	Energy consumption (kJ)
2.3 GHz	504	675.5
2.0 GHz	510	643.7
1.5 GHz	541	613.3
1.2 GHz	573	608.1

TABLE 7.4: NEMO remarkable frequency raw results

the conclusion that the NEMO application is a balanced application, with a tendency to memory-bound behavior.

From the previous chart, moreover, we selected also the frequency to be used by BDPO as lower and upper frequency. Indeed, it is noticeable that for 1.5 GHz, the energy gain is almost maximal with a reasonable execution time loss, while, for 2.0 GHz, there is a substantial energy saving, around 5%, with less than 1% performance degradation. Table 7.4 shows energy and time values for cited frequencies, together with maximal and minimal ones.

IPC profile

We then studied the NEMO IPC profile in order to better understand its run-time behavior and choose an appropriate threshold to be set in the BDPO tool configuration. Figure 7.8 presents the IPC line for one NEMO run with the already specified configuration, with the BDPO profiler polling period set at 10 milliseconds.

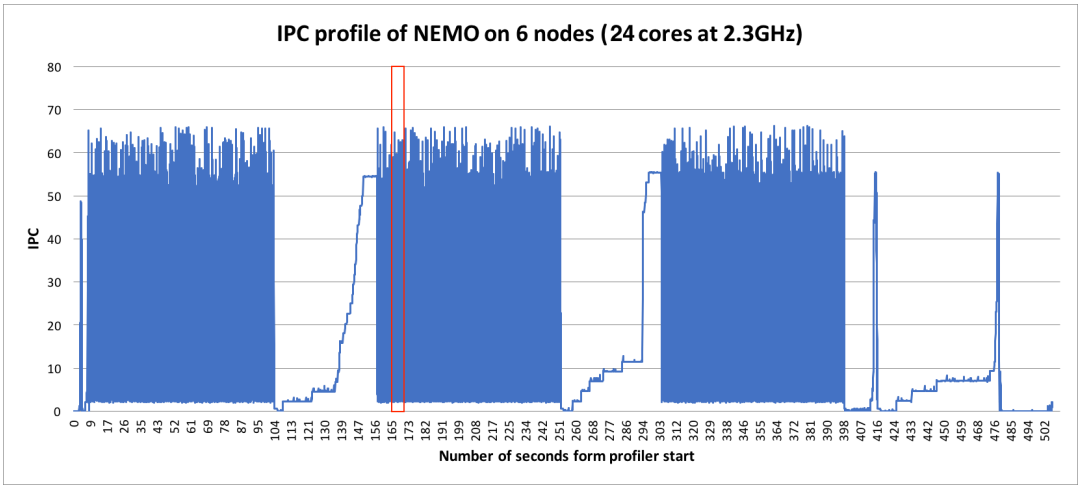


FIGURE 7.8: NEMO IPC profile

As before, the chart shows the IPC profile on one compute node, while for this type of job, 6 different nodes where involved in the computation. One important thing to say before analyzing this profile is that the NEMO behavior was different on each node. In other words, the NEMO application has an unbalanced workload distribution. This functioning is probably due to the nature of the computation: the module we choose to work with, in fact, tries to simulate an oceanic whirlpool. The mathematical equations involved in the simulation are then different in terms of complexity depending on the domain region. Surely, different processes are in charge of the computation of different regions. This means that it is very likely that some nodes have to solve complex computations than others. We saw this difference mainly from the IPC values, that were higher on some nodes on lower for others.

Apart from the latter fact, all the 6 different IPC profiles were perfectly synchronized on the computing and checkpoints phases. This IPC profile, in fact, clearly shows some NEMO characteristics: we can easily distinguish between compute phases and checkpoints. The computing phases are defined by a really variable IPC, while, as it is possible to see from the chart, at the beginning of checkpoints the IPC drop down close to zero. Then, following a step pattern, the IPC slowly raise, till a new computing phase begin.

Why the IPC drops down to zero and then slowly returns up? Because of C-states and process synchronization. We recall that a checkpoint is a heavy data write on disks. Processors, in general, are not responsible of I/O operation, so during a checkpoint, they enter deep idle states, accordingly to section 3.3.2. We recall that with our setup, described in section 4.2.1, we activated idle states, but preventing the system to enter deeper C-states then C1. During C1 state, no instructions are fetched by the execution pipeline, so we found an IPC really close to zero. One important consequence of C-states is that they are really energy safe, performing clock and power gating. When a core goes to C-state, it consume surely less than when it is operating at the minimal frequency. This means that it is not possible to gain energy with DVFS when the core goes into C-states.

Each process is in charge of its part of I/O operation, so that when it

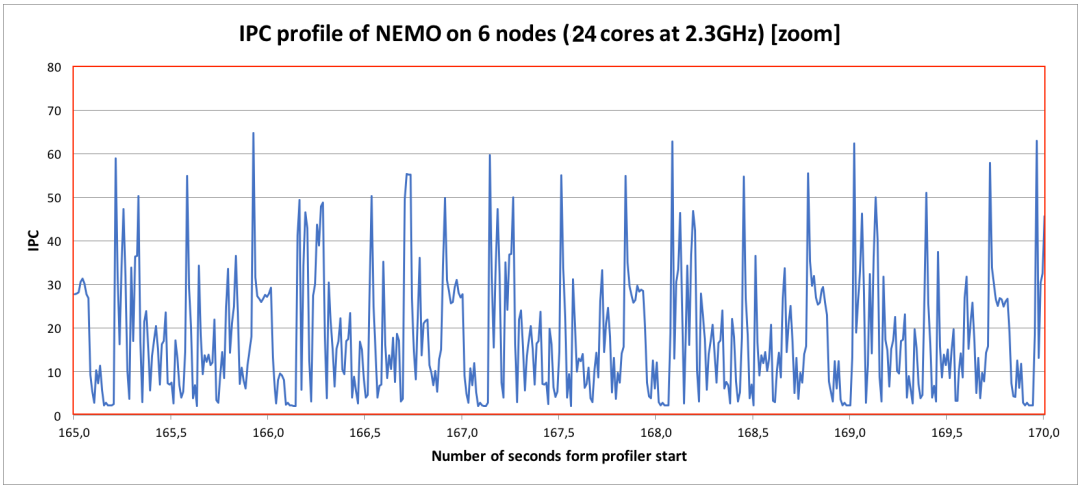


FIGURE 7.9: NEMO IPC profile (zoom)

finishes, it starts to synchronize with the others, with an MPI active wait. This causes the raising pattern during the checkpoint. At the end of the execution, a final write on disks is done.

Figure 7.9 represents a zoomed views of the previous chart, more precisely a 5 seconds slice highlighted by the red box during the second compute phase. As we can see form the chart in the figure, it is clearly recognizable a periodic patter in the IPC profile. This is due by the previous described iterative approach of NEMO. The IPC is not stable, that is to say that there are a lot of variations, with repeated peaks, followed by drops due to MPI communications and memory accesses; however, it is a much more stable profile then the one studied for the WRF application.

We then choose an IPC threshold of 12.5, in order to avoid the very fast and little changes below 10 IPC.

BDPO results with NEMO

In order to test BDPO with the NEMO application and to determine whether its energy consumption could be decreased, we ran NEMO with low frequency set at 1.5 GHz and high frequency at 2.0 GHz, polling period set at 10 milliseconds and with the NEMO usual configuration.

In order to be able to reproduce our measures, we performed 31 runs with all our possible NEMO use cases: the first is the one with BDPO activated, the second was with the frequency governor set to `performance`, the third was with `ondemand` frequency governor, the fourth with the `userspace` governor and a fixed frequency at 1.5 GHz and the last one with the same governor but a fixed frequency to 2.0 GHz. The last two configurations were necessary in order to understand the quality of BDPO mechanics.

Once completed all runs, we computed the average energy consumption and execution time for each groups of NEMO execution. Then, we compared the performances obtained with BDPO with the ones obtained with the two previously mentioned reference contexts. The results are sum up in tables 7.5 and 7.6.

Raw results	Execution time (s)	Energy consumption (kJ)
NEMO with BDPO	521	632.4
`performance` governor	516	689.3
`ondemand` governor	519	713.9
1.5 GHz	539	613.9
2.0 GHz	510	643.4

TABLE 7.5: NEMO application raw results

BDPO relative results	Execution time (s)	Energy consumption (kJ)
`performance` governor	+ 1.00%	- 8.25%
`ondemand` governor	+ 0.42%	- 11.42%
1.5 GHz	- 3.32%	+ 3.01%
2.0 GHz	+ 2.17%	- 1.70%

TABLE 7.6: NEMO application relative results

Table 7.5 shows the raw values for the BDPO case and all the references. Table 7.6 presents instead the percentage relations between the results obtained with BDPO (the first line of the first table) with the other cases. For example, looking at the first line of table 7.6 that is the one referring to the `performance` governor, the most used one in the HPC field, we can see that the test performed with our tool active reaches on average substantial energy saving, around 8%, degrading the performances of only 1%.

Even higher is the save of energy with respect the `ondemand` governor, where BDPO saves more than the 11% of energy facing a performance degradation close to zero.

Comparing the runtime frequency scaling of BDPO with the two fixed-frequency cases, we can see that using all the time the lower frequency of 1.5 GHz we save 3% more energy than BDPO but having around 3% more of performance degradation (this result is in line with the memory-bound behavior we exploited before in this chapter). Using, on the other hand, all the time the higher frequency the NEMO application is 2% faster than BDPO but it use almost 2% more energy.

As a result, for applications such as NEMO, it seems really possible to optimize the job energy consumption thanks to BDPO.

8 Conclusions and future work

We started this thesis by taking into account the problem of the increasing energy consumption on modern supercomputers. Today's HPC systems have reached the computing power of 100 Peta-Flops, consuming around 15 mega-watts. Research centers, system integrators, and chip makers are driving the effort to hit the Exascale goal in the upcoming years, that roughly means to build a supercomputer with ten times more computing power than modern machines, turning over the 1 Exa-Flops. This also means, by estimation, to consume ten times more energy. Of course, it is not possible for cost and energy supply reason to have a supercomputer that consumes 150 mega-watts. In order to build a sustainable system, we would expect to budget around 20-30 mega-watts of power consumption. This simple example clarifies the need of constant research on energy consumption of HPC systems, both in term of hardware and software consumption.

8.1 Objectives and findings

We then come with the idea of a software solution that aims to dynamically reduce the power consumption of a parallel job, at runtime. Our initial objectives were then to:

- Reach real HPC application optimization: we did not want to test our tool only on well-known benchmarks where we precisely know what to expect in term of application behavior, but we targeted real cases used in nowadays market by HPC customers, in realistic operating conditions.

- Optimize applications without change of their source code: this means to have a non-intrusive approach, so that the system must be able to optimize the energy consumption of any job submitted on a cluster, without depending on the actual implementation of the job application. This also allows optimizing the consumption on already existing applications, without asking programmers to modify the already written code.

- Do not affect application performances: we wanted to create a solution that causes no degradation, or at least as less as possible and the closest to zero. This point is very important in the framework of HPC systems, while they are built to reach the highest performances possibles.

- Have no information on target application: eventually, we want that our system will be able to be completely automatic. In other words,

no profiling of the target and tool parameters tuning are necessary to improve the energy consumption of the machine.

With all those objectives in mind, we designed the Bull Dynamic Power Optimizer (BDPO), a system that is able to compute some real-time metrics to understand the compute node behavior. In particular, as main metric we used the Instructions Per Cycle (IPC), that is a per-core metric aiming to represent the workload on the core. Based on that, BDPO takes some decisions figuring out the phase in which the system is going in or is coming from. To every decision corresponds an action, that is to apply the DVFS technique on all processing cores on the machine CPU: if the application is in a memory phase or message passing phase, that usually correspond to a low IPC value, BDPO puts on the node a low operating frequency, in order to consume less energy. On the contrary, if the program is entering a computing phase with a high IPC, it sets the system operating frequency to a high frequency permitting to save energy thanks to a faster execution.

After the BDPO development process, we tested it on some real HPC applications. We took into account three different applications: GROMACS, WRF, and NEMO. Those three differs by the application domain, models involved, architectures and programming languages. We also had different results for each one of them: for GROMACS we find out during its study that it is a CPU-bound application and the frequency that best fit in this case is the highest available operating frequency. We then didn't apply the BDPO tool on the GROMACS application. We did it, instead, on the other two HPC programs. With WRF we find out a limitation of the IPC metric: in this case, the too high variability of the metric did not allow to reach an energy gain, while we introduced a performance overhead of less than 5%. For NEMO, instead, we reached to gain from 8% to 11% of energy consumption with respect to the most used references cases, having a maximum performance loss of 1%.

8.2 Future work

All those results are really important: first, we validate our software solution on real HPC application, demonstrating that is actually possible to gain energy using our approach. We saw that in the worst case it is possible to introduce some performance loss, but they are still reasonable, under the 5%. Moreover, we learned a lot from the test we performed: we find out that an important next step is to add more metrics to our set. As we saw from the WRF case, the IPC by itself could be insufficient in some case. A deep study has to be performed on memory metrics, I/O metrics and on power/energy metrics in order to be able to better understand the application behavior at runtime.

Furthermore, another important aspect to be enhanced into BDPO is a more intelligent decision maker, with the possibility to take a decision based on the metric history rather than the last value retrieved. On one hand this will also help to do not take some unnecessary actions, drastically reducing

the BDPO overhead; but on the other hand, a too complex logic could introduce more weight in the trigger mechanism, affecting performances.

Based on our description, in order to test BDPO on real applications we had to perform some studies and profiling them, that is in contrast with the last objective we presented early in this chapter. To be able to work, BDPO needs at least three parameters, that are the values of the high and low frequency to be used for DVFS, and the metric threshold for the trigger mechanism. So far, those three parameters have to be given and are fixed at the BDPO start. Future development is necessary to reach the point where BDPO will auto tune those parameters values at runtime.

In conclusion, in this thesis we show that in the context of real HPC applications, relying on real-time metrics monitoring and the DVFS technology, without modifying in any way the target application, we reached up to 11% the energy savings with less than 1% performance loss.

Bibliography

[1] H. S. M. M. Erich Strohmaier, Jack Dongarra, "Top500 supercomputer list website," *https://www.top500.org/lists/2017/06/*, June 2017.

[2] N. Gholkar, F. Mueller, and B. Rountree, "Power tuning hpc jobs on power-constrained systems," in *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 179–190, Sept 2016.

[3] N. Triquenaux, *Energy Characterization and Savings in Single and Multiprocessor Systems : Understanding how much can be saved and how to achieve it in modern systems*. Theses, Université de Versailles Saint-Quentin-en-Yvelines, Sept. 2015.

[4] J. Eastep *et al.*, "Global extensible open power manager: A vehicle for hpc community collaboration on co-designed energy management solutions," *Lecture Notes in Computer Science*, vol. 10266, 2017.

[5] B. Unni, N. Parveen, A. Kumar, and B. S. Bindhumadhava, "An intelligent energy optimization approach for mpi based applications in hpc systems," *CSI Transactions on ICT*, vol. 1, pp. 175–181, Jun 2013.

[6] "Mont-blanc, european approach towards energy efficient high performance," *http://www.montblanc-project.eu/home*, August 2017.

[7] T. F. J. C. B. W. J. Kelly Livingston, Nicolas Triquenaux, "Computer using too much power? give it a rest (runtime energy saving technology)," *Computer Science - Research and Development*, May 2014.

[8] B. P. J. C. B. W. J. Nicolas Triquenaux, Alexandre Laurent, "Automatic estimation of dvfs potential," *Green Computing Conference (IGCC), 2013 International*, June 2013.

[9] A. G. W. J. Jean-Philippe Halimi, Benoît Pradelle, "Forest-mn: Runtime dvfs beyond communication slack," *Green Computing Conference (IGCC), 2014 International*, Nov. 2014.

[10] C. A. Mack, "Fifty years of moore's law," *IEEE Transactions on Semiconductor Manufacturing*, vol. 24, pp. 202–207, May 2011.

[11] "Intel® 14 nm technology," *https://www.intel.com/content/www/us/en/silicon-innovations/intel-14nm-technology.html*, June 2017.

[12] "Introducing titan | the world's n.1 open science supercomputer," *https://www.olcf.ornl.gov/titan/*, Oct. 2012.

[13] S. Park, Y. Kim, B. Urgaonkar, J. Lee, and E. Seo, "A comprehensive study of energy efficiency and performance of flash-based ssd," *Journal of Systems Architecture*, vol. 57, no. 4, pp. 354 – 365, 2011.

[14] A. K. C. Fulya Kaplan, Jie Meng, "Optimizing communication and cooling costs in hpc data centers via intelligent job allocation," *Green Computing Conference (IGCC), 2013 International*, June 2013.

[15] R. Mahdavi, "Liquid cooling v. air cooling evaluation in the maui high performance computing center," *U.S. Department of Energy Federal Energy Management Program*, July 2014.

[16] "Advanced configuration and power interface specification," *http://www.acpi.info/index.html*, July 2014.

[17] "Power management states: P-states, c-states, and package c-states," *https://software.intel.com/en-us/articles/power-management-states-p-states-c-states-and-package-c-states*, April 2014.

[18] A. Mazouz, A. Laurent, B. Pradelle, and W. Jalby, "Evaluation of cpu frequency transition latency," *Computer Science - Research and Development*, vol. 29, pp. 187–195, Aug 2014.

[19] "Slurm workload manager," *https://slurm.schedmd.com*, November 2016.

[20] "Cpu frequency scaling," *https://wiki.archlinux.org/index.php/CPU_frequency_scaling*, July 2017.

[21] "Cpu frequency and voltage scaling code in the linux(tm) kernel," *https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt*, August 2017.

[22] "Perfmon2/libpfm4, improving performance monitoring on linux," *http://perfmon2.sourceforge.net*, Nov. 2016.

[23] "Papi - performance application programming interface," *http://icl.utk.edu/papi/*, July 2017.

[24] H. McCraw, J. Ralph, A. Danalis, and J. Dongarra, "Power monitoring with papi for extreme scale architectures and dataflow-based programming models," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 385–391, Sept 2014.

[25] S. Pandruvada, "Running average power limit," *https://01.org/blogs/2014/running-average-power-limit-—rapl*, June 2014.

[26] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, pp. 41–50, Jan 2003.

[27] "Multiplying matrices using dgemm," *https://software.intel.com/en-us/node/529735*, July 2017.

[28] A. Mazouz, D. C. Wong, D. Kuck, and W. Jalby, "An incremental methodology for energy measurement and modeling," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, (New York, NY, USA), pp. 15–26, ACM, 2017.

[29] "Gromacs hpc application - fast, flexible, free," *http://www.gromacs.org*, June 2017.

[30] "Wwather research and forecasting model," *https://www.mmm.ucar.edu/weather-research-and-forecasting-model*, August 2017.

[31] "Nemo - community ocean model," *https://www.nemo-ocean.eu*, August 2017.