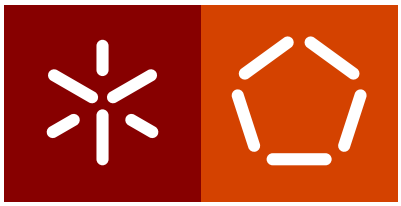


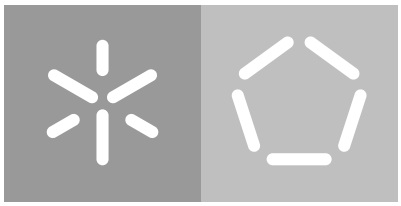
Universidade do Minho
Escola de Engenharia
Departamento de Informática

David Luis Moniz Branco

Optimization in code generation to reduce energy consumption

November 2018

This work is funded by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT - Foundation for Science and Technology, within projects: FCOMP-01-0124-FEDER-020484, FCOMP-01-0124-FEDER-022701, and grant ref. BI4-2014_GreenSSCM-38973_UMINHO.



Universidade do Minho
Escola de Engenharia
Departamento de Informática

David Luis Moniz Branco

Optimization in code generation to reduce energy consumption

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Pedro Rangel Henriques

November 2018

ACKNOWLEDGEMENTS

I want to thank to Prof. Pedro Rangel Henriques for his consideration, dedication, teachings and guidance that came since the first day of the Computer Science Degree.

I am also grateful to Dr. Rui Pereira, Prof. João Saraiva, Prof. Jácome Cunha and other members of the GreenSSCM project team for their support, passed knowledge and always pertinent suggestions.

I also want to thank all my colleagues who have always provided assistance and great moments during the journey. Miro, Carção, Daniel, João, Hugo, Nuno, Marina and the remaining group that was part of the Infobiz family.

Lastly, I would also like to thank my family for their support over time and for the opportunity they gave me to carry out this work.

ABSTRACT

In recent years we have witnessed a great technological advance accompanied by an equally impressive increase in energy consumption, causing problems of both financial and environmental order. In order to counteract this tendency, Green Computing emerges with a number of measures for a more efficient use of computing resources without a great loss of performance.

This essay is a study of several elements of Information Technology analyzed from the point of view of energy efficiency. With special emphasis on microprocessors, modern compiler design, development tools and optimization of code generation, a wide range of information is gathered on very relevant subjects through perspectives still not very considered by the community in general.

Also presented are two experimental studies that analyze the optimization of generated code for a set of benchmark programs in several programming languages with the aim of apraise the otimization impact on improving their energy consumption efficiency. A software measurement framework was also developed that, together with the methodologies presented in both studies, allows obtaining very precise and pertinent results for analysis. Finally, a ranking was produced for 18 development tools, considering the execution time and energy consumption of the executables generated through their compilation profiles.

This study also intends to contribute to an energy efficient technological advancement. All the work developed here may also serve as motivation so that these and other aspects of Information Technology may be seen through a greener perspective.

RESUMO

Nos últimos anos temos assistido a um grande avanço tecnológico acompanhado por um aumento igualmente impressionante do consumo energético, provocando problemas quer de ordem financeira quer de ordem ambiental. Com o intuito de contrariar essa tendência, surge o Green Computing com várias medidas para uma utilização mais eficiente dos recursos computacionais sem grande perda de performance.

Esta dissertação apresenta um estudo relativo a diversos elementos das Tecnologias de Informação analisados do ponto de vista da eficiência energética. Com especial destaque para microprocessadores, conceção moderna dos compiladores atuais, ferramentas de desenvolvimento e geração de código otimizado, é aqui reunida uma vasta gama de informação sobre assuntos bastante relevantes segundo perspetivas ainda pouco consideradas pela comunidade em geral.

São também apresentados dois estudos experimentais que analisam a otimização do código gerado para um conjunto de programas benchmarks em várias linguagens de programação com o objetivo de compreender o impacto das otimizações no sentido de melhorar a eficiência energética dos programas compilados. Foi também desenvolvida uma framework de medição por software que em conjunto com as metodologias apresentadas em ambos os estudos permite a obtenção de resultados bastante precisos e pertinentes de análise. Por último é elaborado um ranking para 18 ferramentas de desenvolvimento considerando o tempo de execução e consumo energético dos executáveis gerados através dos seus perfis de compilação.

Este estudo pretende assim contribuir para um avanço tecnológico energeticamente mais eficiente. Que todo o trabalho aqui desenvolvido possa também ele servir de motivação para que estes e outros aspetos das Tecnologias de Informação possam ser vistos através de uma perspetiva mais ecológica.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

B

BAT Build Automation Tool.

C

CLBG Computer Language Benchmarks Game.

CPU Central Processing Unit.

G

GCC GNU Compiler Collection.

GPU Graphics Processing Unit.

GREENSSCM Green Software Project for Space Control Mission.

H

HTML Hypertext Markup Language.

I

ICT Information and Communication Infrastructure.

IDE Integrated Development Environment.

IT Information Technology.

J

JIT Just-In-Time.

O

OS Operating System.

R

RAM Random Access Memory.

RAPL Running Average Power Limit.

V

VM Virtual Machine.

1

INTRODUCTION

This project was initially funded by a research grant, under the Green Software project for Space Control Mission (GreenSSCM[1]) / University of Minho, financed by European Regional Development Fund (ERDF) through the Programa Operacional Regional do Norte.

1.1 CONTEXT AND MOTIVATION

Currently we live in a period in which technology evolves very quickly and the number of those who use it, causing the associated energy consumption, reaches very high values in financial and environmental terms (Guelzim and Obaidat, 2013; Zhang and Ansari, 2013).

So there is a strong concern and an increasing need to reduce energy consumption in all the information and communication infrastructures (ICTs). Together with these factors emerged Green Computing, also the so-called Green Technology, which aims precisely at using computing resources more efficiently while maintaining or increasing overall performance (Harmon and Auseklis, 2009). This paradigm has already more than three decades and directly covers the entire ICT infrastructure (Harmon and Auseklis, 2009). Even though this paradigm exists for some time, it never received proper attention, which is evidently one of the main reasons for the problems and limitations associated to the environmental impact of the IT industry (Harmon and Auseklis, 2009). This area is very wide, then the question arises how can we reduce this consumption. Among the various possibilities, the chosen approach will be to study the reduction of energy consumption during program execution. This possibility raises several issues, namely: how can we reduce the energy consumption without affect the reliability of the application?; how does the energy consumption relates to all other aspects of the generated code?; what is the current stance of the compilers on this matter?; what compilation techniques can help us contradict this presented tendency?; what is the actual impact on the compiler's generated code and the respective optimizations in terms of energy consumption?; what are the development tools that offer the best results in this strand? These are few of the major concerns that will be

1 http://visionspace.dnsdynamic.com/GreenSSCM

discussed in this Master's thesis.

Beyond the academic component of this project, there is also a great personal engagement. It is very motivating to know that I can contribute to the solution to a real problem that tends to increase (Koomey, 2011). The choice of this topic took in consideration these aspects of motivation as well as an opportunity to apply the knowledge obtained over five years, thus enriching the education in theory and practice.

1.2 OBJECTIVES

The main focus of this thesis is to carry out, in a clear but detailed manner, a study about the current role of Green Computing in technology, by gathering a great number of related elements, in terms of hardware and software. It will also explore how do they relate to themselves and to other surrounding elements, discovering their impact on energy consumption, and understanding how the consumption can be optimized. This work will emphasize the matters concerned with processors, compilers, programming languages and development tools. A historical, current and future context will also be carried out whenever it is considered relevant to the intended analysis. Nonetheless, it is also intended to analyze other important strands for each element beyond the perspective of Green Computing.

In particular, it is expected that at the end of this dissertation the following goals will be achieved:

- Clear perception about the role, importance and potential of Green Computing in Information Technologies;
- Investigation of the weight of energy consumption on different types of hardware and detailed analysis of the role of the most relevant element;
- Investigation of the importance and potential of using software (with the main focus on compilers) to attain energy efficiency;
- Comprehension of the hardware-software-energy triplet in order to enhance significant improvements;
- Study of relevant software development tools (with the major focus on IDEs) and the performance of the executables they generate by default;
- Definition and application of study methodologies that allow to apply the knowledge acquired for concrete case studies with relevant elements;

- Obtaining results that allow not only to draw relevant conclusions in the scope considered but also to be used in other areas of analysis.

1.3 STUDY PLAN

Firstly, this report starts by defining the concept of Green Computing: what is its origin, its evolution and how relevant it currently became. Furthermore, it presents its goals and most relevant solutions, picturing what can technology benefit from this paradigm. This dissertation then discusses hardware manufacturers, more precisely the most renowned microprocessors manufacturers, giving an overview of the industry's current position on Green Computing, from past to the current days. With all these facts it can portray the potential of the microprocessors in reducing the energy consumption on Information Technologies.

It is revealed that it is equally important, and essential, to study the design of modern compilers and how do they relate to those previously mentioned elements. The aim of such study is to understand concretely how they act and what properties guarantee the code they generate, what considerations do they have regarding the topic under consideration, which optimization techniques currently stand out and what effect do they have on the energetic consumption of the generated code.

Furthermore, the dissertation presents an experimental study to demonstrate the acquired knowledge and understand, for each specific use case, how do these elements behave and analyze in detail all relevant features for real use case scenarios.

Particularly, the study observes what is the impact of the GCC optimization suites on the energy consumption of running programs. This subject allows to integrate, in experimental form, many of the aspects analyzed such as the hardware components, modern compilers and programming languages. This being presented under the perspective of the energy consumption and all around related concepts. The study shall select the most well known and commonly used programming elements to increase the relevance of the obtained results.

After concluding the previous study, it is also relevant to discover how the analyzed practices are integrated with the developer process, and what is the impact on the code they produce. All these questions motivate the conduction of another experimental study that conciliates the acquired knowledge with the introduction of new elements that are indispensable in the daily routines of a programmer (software development environments, benchmarks, compilation parameters, among others).

Both experimental studies will require very rigorous measurements of energy consumption and related concepts, treatment of results that must be displayed in appealing formats in order for them to clearly become an asset for the intended analysis, study of relevant programs and prone to be compared effectively, among many other competences which need to be integrated into the desired objective. Although all the vital cooperation between heterogeneous elements and the aggregation of knowledge acquired from multiple areas is in fact an additional and quite demanding factor of difficulty, it is also a motivating factor that leads us into moving in that direction.

1.4 DOCUMENT STRUCTURE

This document contains six chapters.

The two sections that make up the State of the Art explain, in a theoretical way, the various issues related to the context and motivation for this dissertation. Section 2.1 highlights the problems that have led to the foundation of Green Computing, defines and describes its goals, presents a review of its evolution over time as well as some important landmarks, and finally reveals the solutions proposed to solve the drawbacks identified. Section 2.2 discusses the heterogeneous utilization of this technology. The evolution of the requirements of manufacturers and the consumers demand over the years is reviewed, in which the importance and growing complexity of microprocessors is described, and finally its energetic impact on a computer system is analyzed. Following, in Section 2.3 some microprocessor manufacturers are studied within Green Computing perspective. It is analyzed what tools customers and developers are provided with, and also what is indicated in the instruction set of some of their latest products regarding energy optimization. Finally, the Chapter 3 examines some interesting features in the design of modern compilers, especially those that focus on energy efficiency issues. Initially it is verified what properties are currently considered ideal by the compilers in code generation. Then, the differences between power and energy are discussed and four energy aspects relevant to hardware and software are also described. Lastly, while maintaining the focus on achieving energy benefits, several optimization techniques are presented in order to obtain greater advantages in the generated code.

Following the survey of all relevant information, there are two chapters referring to experimental studies.

Chapter 4 examines the impact of GCC optimization levels on energy consumption during program execution. After exposing some interesting case studies, the elements that are specifically part of the study are described and how it will proceed. Finally, the results obtained and the consequent conclusions are presented and discussed. In turn, Chapter 5 examines the impact on energy consumption of programs compiled using Integrated Development Environments. After a brief introduction, some observations are made regarding this kind of tools, namely what are their main functionalities, what are the main advantages/disadvantages they offer and the differentiating aspects in relation to the existing alternatives. Next some more contextualization is made through the presentation of the project where the benchmarks will be extracted and which type of characteristics the compilation profiles have. After completing the more theoretical part, the selection and description of the elements under study starts and the applied experimental methodology is defined. Finally, we present and discuss the results obtained through an exhaustive analysis of the various aspects considered.

Lastly, in Chapter 6 a conclusion is devised focusing the main considerations of the present work.

2

PROCESSOR VENDORS AND GREEN COMPUTING

2.1 GREEN COMPUTING

Green Computing appeared to address the problems associated to IT industry policies concerning their sustainability and impact on the environment. In this section will be explored various topics related to Green Computing, including:

SUBSECTION 1: Some of the problems that led to its arrival are explained in concrete;

SUBSECTION 2: A definition and description of some of its goals is presented;

SUBSECTION 3: A description of a few landmarks and its evolution is reviewed;

SUBSECTION 4: Its importance and what measures and solutions have been proposed are highlighted.

2.1.1 *Advent of Green Computing*

The main target of the IT industry since the beginning was the processing power of their products. Although this objective has been achieved surprisingly well and incredibly fast, the proper attention was not given to other important factors like cooling, power consumption and space for data centers since they were considered always affordable and accessible (Harmon and Auseklis, 2009).

As a result of rapid development as well as of the high demands, some facts soon evidence that this approach was problematic and limited, as said in (Harmon and Auseklis, 2009):

INCREASING ENERGY COSTS

It has been estimated, in an article in Computer Weekly journal (Computer Weekly, 2006), that the office equipment during the year 2006 was responsible for 15% of the power consumed in all United Kingdom. It was also estimated that in 2020 the consume would be about 30%, with computer equipment responsible for about two-thirds of this energy consumption (Barnatt, 2012).

For Data Center Servers this scenario is even worse because they use 50 times the energy per square foot as an office does (Roy and Bag, 2009).

The energetic consumption of data centers worldwide doubled from 2000 to 2005 and these values tend to increase in the future as well has the number of data centers (Roy and Bag, 2009).

INCREASING COOLING REQUIREMENTS

To prevent overheating of their IT equipment the companies usually resort to air-conditioning equipment, where the more powerful machines need even more attention and support. This necessary measure entails more expenses for companies and every time they think about expanding their equipment they must think too in extra cooling costs. The energy required to power and cool the United States servers in 2005 was about 1.2% of all electrical energy consumed in the country (Roy and Bag, 2009).

At worldwide level and according to Gartner (an American IT research and advisory firm) in 2010, about half of the companies listed in Forbes magazines top 2000 had more expenditure in power consumption of their servers than with hardware (Roy and Bag, 2009). Nowadays circa 10 per cent of the average IT budget is for energy costs but could rise to 50 per cent in several years, which is unsupportable for most companies (Roy and Bag, 2009).

RESTRICTIONS ON ENERGY SUPPLY AND ACCESS

Sometimes in some regions the energy necessary for consumption and cooling of IT equipment reaches such high amounts that the power supply is not able to provides or requires it even higher costs. Because of this companies such as Yahoo, Microsoft, Google established or migrated their large data centers for colder regions of the world (e.g. in Northern Europe) or near hydroelectric power station (e.g. Columbia River in the USA) where they have direct access to low-cost energy to mitigate the problem (Harmon and Auseklis, 2009).

INCREASING EQUIPMENT POWER DENSITY

Usually in IT companies whenever it is necessary to improve the performance of their servers the option taken is to install new equipment with more processing power and memory. As a result of the rapid development of IT equipment, it can take up less area (in some cases more than 70 %) than the previous machines but consume more energy. As a consequence this increase of processing capacity and energy consumption per square meter, is considered the main reason of the growing power density of data centers. This density has increased 10 times more from 91.5 watts per square meter in 1996 to over 1219 watts per square meter in 2007, a trend that is expect to continue its upward spiral (Harmon and Auseklis, 2009).

RAPID GROWTH OF THE INTERNET

The Internet has a huge diversity and amount of content such as social networks, newspapers, videos, e-commerce or online gaming and is increasingly easy their access at high speeds and in diverse devices.

The great success of the Internet also led to a change in the interaction paradigm between companies and institutions and their users. As for example banking and government institutions relegate most of the services and features to their sites and notify users by electronic means making it almost unnecessary interaction in their service counters. Data security reasons require to make backups of existing information. All these factors are increasing the use of the Internet every year (about 10% per year) thus causing a large increase in the size of data centers and in memory capacity of these to meet the growing demand (Harmon and Auseklis, 2009).

GROWING AWARENESS OF IT'S IMPACT ON THE ENVIRONMENT

From the green perspective, computers and their manufacture are not currently very effective. The normal production of computers makes use of toxic materials such as mercury, cadmium and lead. According to green experts a computer normally contains between 1.8kg and 3.6kg of lead alone making them along with other electronics, responsible for two-fifths of all lead in landfills (Roy and Bag, 2009).

Having said that, it is natural that in 2010 only 2 of the 18 manufacturers listed in the Greenpeace Guide to Greener Electronics have been listed as reasonably green rating, highlighting the negative Nokia and Sony Ericsson (Barnatt, 2012).

The increasing use of IT equipment caused that in 2007 there were significantly 44 million servers consuming 0.5% of all electricity in the world. The energy consumed has a direct impact on the environment because it is proportional to the amount of carbon emissions and only these servers are responsible for the emission of 80 metric megatons of CO_2. It was estimated that this value can grow more than 11 per cent a year to around 340 metric megatons by 2020, values that overcome more than three times the amount emitted by Czech Republic (112 Mt), more than double the amount emitted by United Arab Emirates (168 Mt), more than Poland (317 Mt) and approaches the values of France (361 Mt) and Australia (373 Mt) in 2010 (The World Bank, 2014; Forrest et al., 2008; Harmon and Auseklis, 2009).

Based on all these issues easily follows that computing is not very environment friendly and this becomes increasingly evident (and even unsustainable) for companies, organizations and public opinion. In order to reverse this situation appeared the Green Computing.

2.1.2 *Meaning and Objectives of Green Computing*

Green Computing, also called Green IT or ICT Sustainability, is essentially the study and practice of efficient and environmentally sustainable use of computers and related resources. A more comprehensive definition is given by Green Computing Initiative (controller of the industry standards EFGCD — Eco–Friendly Green Computing Definition) defining Eco-Friendly Green Computing as the study and practice of the design, development, implementation, utilization and disposal of IT infrastructure efficiently and effectively with low or zero impact on the environment while reducing operating costs. Combines a wide variety of aspects of the whole computing and also has practical effects for other related fields (e.g. green chemistry). Some of its main objectives are (Roy and Bag, 2009; Jindal and Gupta, 2012; Harmon and Auseklis, 2009):

- Reduce the use of hazardous materials;
- Use new energy-generation techniques (e.g. sun, wind, water, sugar);
- Maximize energy efficiency during the product's lifetime;
- Minimizing waste from manufacturing and throughout the supply chain;
- Promote recyclability or biodegradability of defunct products and factory waste;
- Study of advanced materials to be used in our daily life;
- Decrease pollution and the environmental impact.

It is an increasingly concern not only for a government or environmental IT organizations but also for business and other industries since they noticed that has benefits at financial level (e.g. cost reduction) and improves the public image of the company (Gingichashvili, 2007).

If the companies, from the financial perspective, can manage any losses using the proper mechanisms for that purpose (for example bank loans and sale of shares), from the public image perspective they are increasingly pressured to adopt more environmentally friendly measures and cannot change that fact without a deeper change in the company policy.

Yearly many entities such as magazines (e.g. Newsweek) or environmental organizations (e.g. Greenpeace) make rankings of brands that are based merely on aspects of Green Computing. These rankings are becoming more widespread and sought after by the general public that is increasingly concerned with this aspect and by the brands which sometimes use them in marketing campaigns.

2.1.3 *Roots of Green Computing*

Although this concept only become more popular over the past 15 years, its original idea arouse in the early 90s. More precisely in 1992 the U.S. Environmental Protection Agency started a voluntary program called the Energy Star, taking advantage of the Environmental protection Agency (EPA) Green Lights program launched to promote energy-efficient lighting, in order to promote and recognize energy-efficient electronic equipment such as climate control equipment, monitors, and other technologies (Harmon and Auseklis, 2009).

Many references to Green Computing were found at USENET posts just after the Energy Star has begun, leading to believe it may have been here origin of the term (Roy and Bag, 2009).

From this program resulted the large scale adoption of sleep mode by consumer electronics allowing a product reduce the amount of energy consumed when after use it automatically switches to sleep mode (Roy and Bag, 2009; Gingichashvili, 2007). This program was then adopted by other countries and allowed products minimize the electricity waste.

In 1998 was founded a nonprofit organization called China Energy Conservation Program (CECP) that was responsible for management, administration, and implementation of the certification for water-saving, energy-conserving and environmentally friendly products. The CECP cooperated with manufacturers, motivating them for producing more efficiently, and convincing customers to a more sustainable choice, participating actively in national and international projects, supporting improvements in energy efficiency and environmental protection and assisting social and economic sustainable development (Gingichashvili, 2007).

The Energy Conservation Center in Japan also had an important role in promotion of energy conservation in the industry, for households and local communities, development of human resources engaged in energy conservation and implementation of national examinations, training and seminars (AEEC, 2015).

2.1.4 *Importance and Solutions of Green Computing*

It is important for companies to maintain and analyze all data gathered from transactions that nowadays are ubiquitous and constant. From click stream data and event logs to mobile call records, from the flow of a topic at social networks to most watched video in the week, everything is tracked and has impact both in businesses and in the environment. The necessity to manage this information makes data warehouses and the sprawling data centers growing all the time and that raises a lot of limitations and problems related, for example, with huge amount of power and cooling (was provided in Section 2.1).

The first approaches from IT industry to solve this problem were quite direct and superficial. Trivial solutions, like more efficient cooling systems, will only alleviate the problem and will only add more hardware, quickly becoming an inefficient and unsustainable solution (Jindal and Gupta, 2012).

The transition to Green Computing implies several optimization strategies and the integration of new approaches and technologies for the different existing processes in companies. According to (Jindal and Gupta, 2012; Harmon and Auseklis, 2009) the five core Green Computing technologies advocated by Green Computing Initiative (GCI) are:

GREEN DATA CENTER

In 2008, according to Gartner,

> Traditionally, the power required for non-IT equipment in the data center (such as that for cooling, fans, pumps, and UPS systems) represented, on average, about 60% of total energy consumption.

Added the fact that spendings on servers, power and cooling and management/administration has increased every year (see Figure 2) and these devices have more than 10 years of operation, a first approach must go through investing in new data centers which are designed to be energy efficient or improve some existing components.

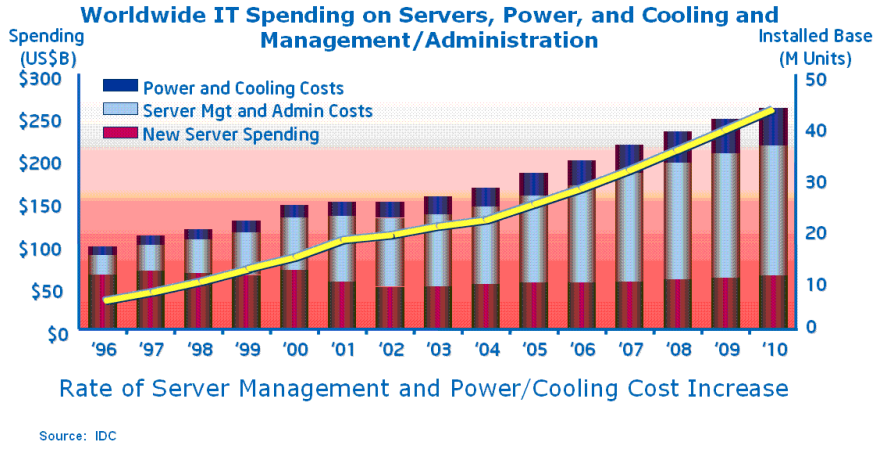


Figure 2.: Worldwide IT spending on servers, power and cooling and management/administration (He, 2008).

Another problem in data centers is the increasing equipment power density. One possible solution begins by a better and efficient heat dissipation strategy like thermal load management. Some examples of practices of this strategy are the variable cooling delivery, airflow management, and raised-floor data center designs to ensure good air flow, more efficient air conditioning equipment, ambient air, liquid heat removal

systems, heat recovery systems, and smart thermostats (Dietrich et al., 2008; Harmon and Auseklis, 2009).

There are also some changes in the servers design that can make them more energy efficient, such as exchange single-core microprocessors for multiple cores (run at slower clock speeds and lower voltages) and the development of dynamic frequency and voltage scaling technologies (allows for microprocessors have a better workload management) (Dietrich et al., 2008).

VIRTUALIZATION

Data center virtualization is one of the most used strategies because it can combine an improvement in the utilization of existing resources with a reduce of costs and human intervention. It covers the areas of server hardware and operating systems, storage, networks, and application infrastructure.

This strategy is used particularly in cases where it is intended to extend the life of older data centers, although there is no room for expansion. One of the main advantages is that it is possible to use less energy regardless of workload level to which they are exposed in relation to standalone servers (for example by pooling applications on fewer servers).

Many operating systems can concurrently run on a same computer by using a hypervisor (a hardware platform virtualization program) that controls the access to the processor and to the memory, giving the illusion that there is only one operating system to the user. This approach can reduce the number of servers, which in turn reduces potential problems associated with them. There is also the concept of storage virtualization, that is analogous to server virtualization, and allows, for example, a reduction in the required disk space as well as better management of storage capacity (Dietrich et al., 2008; Harmon and Auseklis, 2009).

CLOUD COMPUTING

Cloud computing has been a great success in recent years. Forrester Research (independent technology and market research company) estimates that in 2011 was spent $25.5 billion in services (at world level), and as a consequence an annual growth of 22% will reach the $160 billion in 2020. It has several well-known advantages as high scalability and easy access, but there are also some benefits related to Green Computing that are not too accentuated. Resource virtualization (enabling energy and resource efficiencies), automation software (maximizing consolidation and utilization to drive efficiencies), pay-per-use and self-service (encouraging more efficient behavior and life-cycle management) and multitenancy (delivering efficiencies of scale to benefit many organizations or business units) are just a few examples, as stated in (Mines, 2011).

POWER OPTIMIZATION

Power management software allows a more personalized management of energy plans per-user/per-machine according to the worldload. It was estimated that it is possible to reduce the cost of energy of a desktop between 22€ and 65€ per month, an amount that is supposed to be even greater for servers. There is currently a significant market for this type of software that includes renowned brands such as 1E Night-Watchman, Data Synergy PowerMAN (Software), Faronics Power Save and Verdiem SURVEYOR. Most products offer Active Directory integration, multiple power plans, scheduled power plans, anti-insomnia features, undervolting and enterprise power usage reporting (Harmon and Auseklis, 2009; Grier, 2009; Bemowski, 2010).

GRID COMPUTING

Grid Computing is a group of independent computing resources that are interconnected together with the same objective. Within this distributed system such independence is revealed in the heterogeneity and geographical distance in their components, which through internet connections will manage to work together.

A practical example of this strategy is a technique named of shared computing, which involves the use of multiple computational resources (usually processing power but also sometimes other resources) donated temporarily by their owners. These resources are elements that normally are provided when have a low or non-existent workload (typically overnight or mealtimes) and thus can minimize the waste of energy which would be inevitable. The great advantage of this technique is that these components when in large numbers and interconnected by networking can match or even exceed the processing power of a supercomputer. Thus through the use of various resources that were idle, can be achieved similar results of an expensive supercomputer without the necessity of this and acting in a better way from the point of view of Green Computing (Strickland, 2008; Foster, 2002).

2.2 MICROPROCESSORS

Today microprocessors — or commonly, the CPUs or just processors — are practically everywhere (Ryan H., 2012). They are found in generic items used everyday as electronic devices or even in products that uses electronics. Microprocessors are used in almost all the modern vehicles and in their accessories (in some cases may contain approximately 50) , home appliances (e.g. refrigerator, microwaves, washing machine, dishwasher, coffee machine), electronic devices (e.g. digital camera, GPS equipment, tablet computer, mobile phone) or even found in various products existing in normal homes (e.g. toys, alarms, thermal sensors, DVD players).

Much has changed in microprocessors since its inception. Since the beginning of its distribution in commercial versions (early 70s) they were seen by companies as a promising device able to boost and lead the technological improvement. In fact, the ambition for better performance as well as advances in computer architecture, IC fabrication processes and design methodologies made microprocessors rapidly evolve (Betker et al., 1997).

2.2.1 *Different concerns in the development over time*

During the last 40 years the focus of microprocessors development was adjusting according to the requirement and necessity of users.

In the early 80s the great requirement of large firms (main customers of microprocessors) was directed towards them functionality to produce the desired results. They were mainly used in supercomputers and mainframes, and characteristics such as processing power, product cost or energy consumption were not considered as main concerns.

In the 90s the development of microprocessors, so that they can extend the market beyond the big companies, also started to have as a new requirement the product price. With the increased demand for personal computers, this aspect also happened to be rated by users. As a result, new users now started assessing the functionality they were buying was worth the investment they were doing.

The functionality of microprocessors having regard to its processing power and yet the cost to the market, was only the main focus for the industry at the beginning of the new century. After 20 years of manufacturing in commercial format, the functionality of the product was well defined and then came a major concern in making them more efficient. This has become apparent with the emergence of laptops with computing capabilities and price comparable to desktop computers.

In recent years, with the growing awareness of energy issues related to IT industry (see Section 2.1) and with the development of products such as smartphones and tablets, it became clear that this focus was no longer sufficient and acceptable. The main objective is no longer to increase their processing power, but while maintaining the functionality of the product and taking into account the price to the market, reduce its energy consumption. The latest smartphones for example already have a very high processing power, making them equivalent or even superior to the desktop computer used at the beginning of this century. Aspects relating to the autonomy or costs of electricity consumption are nowadays more relevant than increase further processing power (which also becomes increasingly difficult).

The evolution of the industry's requirements over time is described in the Figure 3.

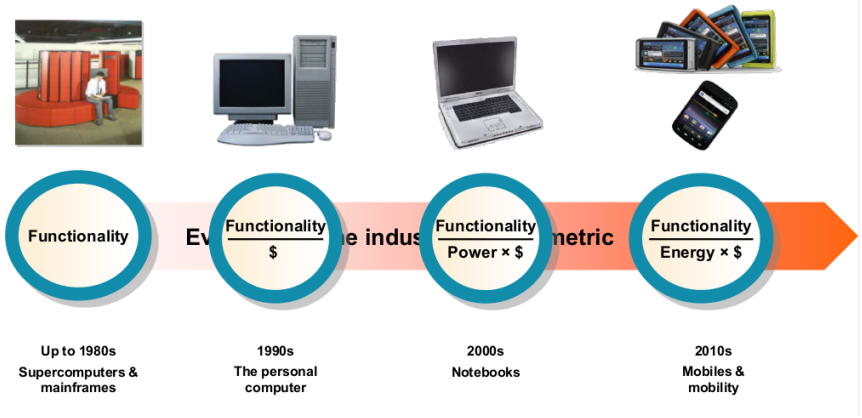


Figure 3.: Industry Changes in Requirements (Davies, 2012).

This new behavior is also stressed by Kathryn McKinley, professor of Computer Science at The University of Texas at Austin, saying (CNS, 2012):

> In the past, we optimized only for performance, (...). If you were picking between two software algorithms, or chips, or devices, you picked the faster one. You didn't worry about how much power it was drawing from the wall socket.

This change in the development focus by manufacturers is then not only because of an increasing awareness of energy issues but also due to a growing search by consumers in products where these limitations are vital, thus forcing manufacturers to look for better solutions.

2.2.2 *Microprocessors as a means for reduction of energy consumption*

Although the basic characteristics that define a microprocessor remain the same from the beginning, the evolution process led them to have multiple execution engines (cores) and complex architectures, as well as a number of extra features such as memory controllers, floating-point units, caches, and media-processing engines.

Being microprocessors the heart and brain of the any usual computer (and also of a huge number of devices used on a daily basis), these new competences make them increasingly responsible for a growing number of tasks (e.g. communicating with new devices). Inevitably, watching the increasing of clock frequencies and transistor count in modern microprocessors also increased the energy dissipation.

All these factors make the microprocessor one of the components with the greatest impact on energy consumption of a computing system. According to the information from Intel

Labs in 2008, (see Figure 4), in servers the processors are even the largest consumers of energy with values between 45W to 200W per multi-core CPU (depending on the type of server and workload) (Minas and Ellison, 2009; Ellison, 2009).

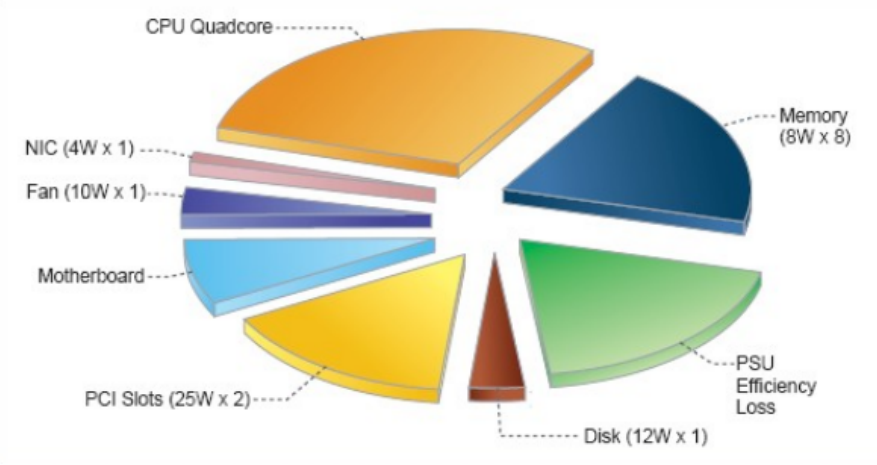


Figure 4.: Server power breakdown by components (Ellison, 2009).

Another important aspect of processors in relation to the other components is that generally are fixed components of a computer over its lifetime. Unlike hard disk storage and RAM which may be upgraded over time, the processor should be necessarily the most appropriate for all the remaining years.

Due to its heterogeneous utilization, computational importance, ability to multitasking, their growing complexity and other factors also aforementioned, microprocessors are arguably the most important component in a computer system. They had a very important role in world's development and will continue to affect the life at several levels of their population. For all those mentioned reasons, the optimization of aspects related with microprocessors is a crucial way to combat excessive energetic consumption of IT industry.

As aforesaid there are already some efforts in this way. For example, the servers from Intel in 11 years have gone from a consumption of 800,000 watts (Pentium) to 10,000 watts (Quadcore Intel Xeon) to process 1.8 teraflops at peak performance. These are impressive figures for such short period of time, but namely in microprocessors, much more can and should be done to reduce the environmental impact of computing (Ellison, 2009).

2.3 MANUFACTURERS OF MICROPROCESSORS

Since its inception to the present day, the market for microprocessors was always seen with great interest by companies. Over the years, more than 30 manufacturers have created their own products promoting a lot of diversity in the solutions, namely in architectures and

way of manufacturing the products. This heterogeneity has been very important for the development of the microprocessor especially in its initial stage. A concrete example of this diversity was the representation of 11 different architectures at the Microprocessor Forum 1992.

Over the last 40 years the market has been changing significantly. It was found that some manufacturers have disappeared, new emerged, some merged, others changed target area or market, and there was also constant changes in the list of most brands market share over time.

Currently among several existing brands, three of them — Intel, AMD and IBM — are distinguished by their importance (either for their longevity, target market, importance in the past or by demonstrating potential).

In the next subsections these companies will be examined in more detail in Green Computing perspective, including its position on this issue, market fields, measures taken already or intended to mitigate the problem, tools available to its customers and developers and also an analysis of what is indicated in the instruction set of some of their latest products.

2.3.1 *Intel*

Intel is a recognized leader and identified as a model by other companies in environmental sustainability. It is the number 1 in several American rankings and also awarded several prizes internationally.

Overall they are being able to minimizing the environmental footprint of operations through measures such as (Economist, 2008):

- air/water/waste programmes such as reducing greenhouse-gas emissions (GHG), water conservation and recycling;
- energy consumption reductions such as energy conservation and renewable power programs;
- dedication to green buildings and IT, and greening operational and supply chain processes.

The company is very active in this direction and have measures covering the entire cycle of its products: development, production, use, and ultimate disposal. Intel works with organizations, customers, and businesses around the world in order to sensitize the industry to voluntarily implement measures to help shape progressive and practical environmental and energy policies (Intel Staff, 2013, 2015a).

The company offers quite extensive software developer's manuals that describe the architecture and programming environment of the Intel 64 and IA-32 architectures. Contains

very detailed guidelines on software optimization to take advantage of performance characteristics (particularly related to energy consumption) of their processors as well as guides to work with performance tools to optimize application performance including compilers, multithreading tools and performance analyzer. They also provide descriptions and suggestions on operating-system support environment, specifically: memory management, debugging, system management mode, interrupt and exception handling, performance monitoring, thermal and power management features, protection, multi-processor support, task management, virtual machine extensions (VMX) instructions, and Intel Virtualization Technology (Intel Staff, 2015b).

Among the many generic measures that disclose to reduce application's power consumption can be found for example reduce to a minimum the number of very common calls (e.g. *gettimeofday()*), repeated events and system calls in order to reduce unnecessary work to a minimum, improve data localization, analyze with some detail the I/O application patterns (e.g. experiencing many cache misses) as well as maximize user time (over system time) (Belinda, 2015).

The company offers power consumption measurement tools (that are used also by Intel engineers) to users who search this functionality. Depending on the case and the desired measurement level, it is possible to choose Frameworks, APIs and another type of tools (10 in total) available for different platforms and programming languages (Belinda, 2015).

Intel also provides useful material for the regular users of its products as guidelines for good practices to adopt while using computing products in general, as well as tools that will warn and increase consumer awareness for some misbehavior.

2.3.2 *AMD*

AMD has published notable values in terms of environmental responsibility and transparency in recent years. The company put into practice measures to reduce environmental impact in their various business processes, particularly from business operations to the supply chain. Through public reports published by the company it is possible to attest its progress in key factors such as reduction of greenhouse gas emissions, use and waste water, waste (non-hazardous). Something remarkable is that the company has planned high and ambitious environmental goals, and they not only has achieved but also exceed them making it clear that they are taking an active role in conservation efforts (Staff AMD, 2014).

As far as the level of improving power efficiency and increasing performance of their products the last results and projections are also admirable. With market interests similar to Intel (more details were provided in previous section) it was reported in 2014, that since 2008 managed a 10-fold energy efficiency improvement in its product line. AMD recently published the targets to be achieved in the entire line of mobile processors until 2020 and

still manage to be higher than those pledged for the last six years. When it was expected that the company would slow down this growing progress (because previously taken measures are nowadays increasingly difficult to perform with the same success) they estimate 25 times higher energy efficiency between 2014 and 2020 (Tirias Research, 2014; Papermaster, 2014). According to the company responsible, these goals will be achieved thanks mainly to three key points (Papermaster, 2014):

- Improvements in intelligent, dynamic power management;
- Heterogeneous-computing and power optimization;
- Future innovations in power-efficient design.

The company provides to customers and users of their products some quite extensive and detailed documentation as reference manuals about several topics (e.g. compilers, software optimization, products manuals), developer guides and also some tools and libraries to help them optimize their software.

2.3.3 *IBM*

IBM mainly develops software and products for their big target market, companies that need more specific products such as business software, mainframe or very optimized disk storage. Remember that IBM works with clients from the most diverse fields, such as Aerospace and Defense, Automotive, Banking, Healthcare, Financial Markets and Telecommunications.

The company has been focused on thematic of Green Computing for over 30, and was even one of the first companies in the IT industry to reduce their environmental impact. The first great measure was in 1971 when the company formally establishes the corporate policy on environmental affairs. Since then has established strong partnerships with governments and environmental organizations (e.g. WWF, Climate Leaders) as well as investigated how to dramatically increase the efficiency of their products. One of the biggest investments made by the company is Project Big Green created in 2008, where IBM invests since then $1 billion per year to meet the energy consumption problem (Wong, 2010). It was several times recognized and distinguished by entities throughout the world for their good work developed, in particular in 2011 was considered the greenest company in the USA by The Newsweek magazine (Newsweek Staff, 2011).

The measures put into practice in the company cover almost all of their own processes, such as reducing the consumption of water and electricity, transport of employees in a more intelligent way and supply chain sustainability management.

Investigating the last line of IBM mainframes (IBM zEnterprise System), it can be detected in their products some of the characteristics that the company considers important

for energy efficient and consumption. Specifically analyzing the IBM zEnterprise 196 model (z196) it is observed that with increased number of available processor cores per server and capacity, and with reduced floor space intend to reduce the energy consumption when doing large-scale consolidations of distributed workloads. To be able to make the z196 extremely energy efficient they resorted to virtualization technologies, advances in microprocessor design, more efficient power conversion and distribution, 45nm silicon technology and more advanced sensors and cooling (high-voltage dc and water cooled). This product also has features such as static power savings mode and query max potential power to reducing wattage and power across the entire data center, as well as resource managers that provides trend reporting and monitoring of energy efficiency for the entire heterogeneous infrastructure. All these measures make the company considers that this mainframes line is the most powerful and energy-efficient System z 'Green IT' server ever (IBM Staff, 2015).

IBM provides quite material to raise awareness in people to adopt best environmental practices when using computer products in general, as manuals with some guidelines, presentations and lectures of their awareness campaigns in other institutions and conferences and even detailed explanations of its products to a more optimal use in energetic terms.

3

COMPILER DESIGN AND ENERGY REDUCTION

Following the growing evolution of technology, modern instruction sets are also increasingly complex. Allying the presented trend with the frequent addition of new instructions, naturally also the manuals of instruction sets become more and more extensive. Note that, for example, Intel's instruction set reference manual (version 064) consists of more than 4700 pages divided into four volumes (N. Hasabnis, 2015) (Intel Corporation, 2017). This fact is even more relevant because the existing approaches to develop architecture specifications are mainly based on manual modeling of instruction semantics. In addition, modern compilers support many different architectures (e.g. GCC over 45 in 2017) which reveals the enormous difficulty that exists to generate code that respects fundamental properties for its accurate creation, full usage, maintenance and respective post-processing tasks such as optimizations (GCC team, 2017).

Compiler design is probably the most mature Computer Science object. Although compilation techniques and paradigms have stabilized over the years, the area is so vast, complex and important that space is never closed for further significant improvements. New features implemented in hardware that need to be harnessed (e.g. dynamic scheduling), optimizations of old algorithms (e.g. extension of optimal code generation through exhaustive search) or even the resurgence of techniques that regained importance (e.g. recursion removal) make it also one of the most challenging areas in Information Technology (Grune et al., 2012).

In a very simplistic way one can define a compiler as a program that receives a text in a given programming language and returns another text in another programming language, without changing the meaning of it. They are like an intermediary agent between programmers and machines whose function is to mediate a translation process between the two parts. Due to their fundamental role in this relationship, they are subject to quite demanding requirements in order to ensure that the process is carried out successfully regardless of the context. Modern compilers must be able to generate always correct code, fully respect the specification of target languages, be able to handle program size programs and still

have an acceptable and suitable build size and time (e.g. linear in the input) (Grune et al., 2012).

Despite the recent interest almost all over the Information Technologies area at least contain the growth of energy consumption in its products, actually some sectors have been for some time making considerable efforts in that direction. One of the most outstanding is the hardware sector, namely hardware architecture, which has been reinforcing this behavior with the prominence of mobile and portable devices. However, without the proper use of efficient software, in practice those benefits are virtually eliminated (Pallister et al., 2013). Therefore, is required a greater effort on the part of the software developers in the accompaniment of this tendency more and more in vogue. On the side of the compilers this effort is already visible since, although energy efficiency is still not considered a property of a good compiler, it is already considered an optimal property of the generated code.

In this chapter are analyzed some interesting properties of compiler and code generation, especially with a focus on energy efficiency issues. In the first section are described the properties nowadays considered optimal in the generated code. In the following is discussed the topic of power and energy and also are described four energy aspects relevant to hardware and software. Thereafter, follows an exposition of five comprehensive domain optimization techniques applied by compilers, but with the primary goal of achieving energy benefits. In the fourth section we present another compilation approach that tries to offer some of the best advantages of compiled and interpreted code. Finally, a balance is taken of what has been discussed and are added some relevant notes.

Most of the content in this chapter was inspired and based on the book Modern Compiler Design (2nd Edition) by five highly respected authors in the area of compilers: Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel JH Jacobs and Koen Langendoen (Grune et al., 2012). That book covers the entire compiler design stack and contains much broader information on aspects that are not considered in this study. It also has a very complete bibliography, as well as other auxiliary material, that nicely complements the discussed subjects. Although the book does not have much detail in the topic intended in this chapter, it has, however, rather solid descriptions and points of view that we consider interesting to share.

3.1 CODE GENERATION

Code generation denotes the phase in which the compiler's code generator transforms an intermediate representation of source code in another one that can be immediately executed by a machine (for example, machine code). The generated code is an object code of some kind of low-level programming language. In the case of a source program written in

a high level programming languages, initially the source is transformed into a lower level language and then is applied the same process mentioned previously. Within the various phases of compilation, code generation can be considered the final stage. Although there is still a possibility for further processing of the code, such as application of optimizations and other processes with similar impact, such tasks can still be seen as part of the phase itself (Grune et al., 2012).

At present, in terms of code generation with optimal quality, modern compilers take into account four properties (Grune et al., 2012):

CORRECTNESS

This is the most important (and at the same time the most defenseless) code generation property. It is obtained mainly through the use of adequate compilation techniques, such as, small semantics-preserving transformation. This approach is preferred to the transformation as a whole from source code to binary object that is admittedly quite complex, demanding and sometimes even incomprehensible.

The most labor-intensive part of a compiler (and also the most error-prone) is precisely the study of which transformations (specially of optimization) can be applied safely, always requiring complex and exhaustive searches in the Abstract Syntax Tree. Compiler writers use the assistance of test suites to ensure that the transformations are correct not only at that time but also in the subsequent steps in the compiler development process.

HIGH SPEED

Improvements in code performance have been synonymous with runtime optimization for many years. Being an attribute so requested all over the industry, is naturally also the main focus of most compiler optimization techniques. For the vast majority of problems, the hardware and software they have at their disposal have enough resources to get better throughput at the detriment of other factors. Consequently, it is obviously one of the main requirements.

Two of the most effective techniques to obtain speed optimizations are in the hands of programmers and therefore outside the domain of compiler design. Choosing a more efficient algorithm or even writing the program (or parts of it) into assembly language are undoubtedly techniques that, despite the inherent disadvantages, allow to obtain quite relevant gains in the program execution speed. For the second situation presented, it has been estimated that it is possible to obtain incredible gains around of 80% - 94% (Roy and Johnson, 1997).

Among the most important techniques for the compiler designer to produce faster code, the following stands out:

- Traditional optimizations, for instance, code transformations that produce faster code as well as the necessary analysis for its correct application;
- Partial evaluation, in which program segments are evaluated even during the compilation phase (constant expressions, etc.);
- Replace code segment jumps by doubling it, allowing not only some improvements but also opening space for new optimizations. Some examples of this technique are unrolling a loop statement by repeating the loop body and inlining function that replaces the function call by the body of the called function.

SMALL SIZE

The code size is no longer a big problem for most users as it used to be years ago. However, for an interesting (and growing) number of technologies it continues to be a determining factor — sometimes even the most important and restraining — within those stated here. Applications (usually mobile) that need to be downloaded as fast as possible or code for embedded systems such as smart cards, household appliances, etc., are two examples that demonstrate the importance of code size for very real day-to-day applications.

Among the several techniques that allow significant reductions in the code size, stand out:

- Procedural abstraction;
- Assorted code compression techniques;
- Aggressive suppression of unused code;
- Threaded code.

LOW ENERGY CONSUMPTION

The low power consumption is already taken by modern compiler designers as one of the four most important factors to appraise the quality of the generated code. This fact becomes extremely relevant to illustrate the global awareness for the problematic in question and also of the commitment of the software in trying to follow the effort of the hardware. The factors directly related to the triplet code-hardware-energy, briefly electrical power management, are two:

- Save energy in order to increase the autonomy and operation time in battery-powered equipment or respective costs with wall-powered computers;
- Protect the various types of hardware (namely processors) limiting peak heat dissipation.

This property will be analyzed in much more detail in the remaining sections of this chapter.

Generally speaking, generated code must respect two essential properties: keep intact the same meaning of the text received originally and still be efficient in terms of resource management at its disposal.

The correctness plays a more important role compared with the remaining properties. The order of importance for the remaining three properties can be considered as presented above, but in practice greatly depends on the context of the problem and the specificity of the solution.

Correctness is achieved through the secure application of good compilation practices. In its turn, the remaining properties are achieved mainly through optimization techniques. Most of the optimization techniques are very inefficient in obtaining improvements simultaneously in the three mentioned properties: high speed, small size and low energy consumption. In practice, they may even be incompatible and it may be necessary to consider a trade-off so that the benefit of one does not over-harm others. Namely, optimizations that have a major impact on reducing execution time are inefficient for code size, and these in its turn, usually generate less energy-efficient programs. But despite this global incompatibility, it is possible to obtain in a relatively simple way good optimizations in the energy consumption when applied techniques oriented to the optimization of the execution time. Afterwards in practice it may depend on a few more factors (such as the specificity of the code, language, or programming paradigm), but for most situations the simple fact that a program spends less time executing means that it spends less energy (Grune et al., 2012) (Pereira et al., 2017).

3.2 POWER REDUCTION AND ENERGY SAVING

Although we all have at least a vague idea of what is potency and energy in electrical terms, it is however necessary to clarify in more detail the difference between both terms since they are sometimes (mis)used interchangeably. It is also important to mention that there are several energy aspects related to the execution of programs. Namely, there are four aspects widely referenced as good indicators for obtaining energy efficiency, increase battery autonomy and the hardware life cycle: peak power, step power, average power consumption and total energy cost.

3.2.1 *Power vs. Energy*

In the book Modern Compiler Design is reported a sidebar that summarizes in a very clear way how they are both terms related and what are their real implication in electrical terms. The title is ”Power and energy — Volts, amperes, watts, and joules” and is transcribed entirely below (Grune et al., 2012):

The tension (voltage) on an electric wire is measured in volts, abbreviated V; it can be thought of as the pressure of the electrons in the wire relative to that in the ground.

When a voltage difference is applied to an appliance, a current starts to flow, which is measured in amperes, abbreviated A; it can be thought of as the amount of electrons per second, and its magnitude is proportional to the voltage difference.

An electric current flowing over a voltage difference produces power, measured in watts, abbreviated W; the power is proportional to the product of voltage difference and current: W = V $\times$ A. The power can be used for purposes like producing light, running a computer, or running an elevator, but in the end almost all power is dissipated as heat. The longer the power is produced, the more energy results; energy is measured in joules, abbreviated J, and we have J = W $\times$ t, where t is the time in seconds during which the power was produced.

If any of the above quantities varies with time, the multiplications must be replaced by integration over time.

On the one hand energy is a more intuitive notion than power, just as length is more intuitive than speed; so power is more easily explained as energy per time unit than vice versa. On the other hand the watt is a more usual unit than the joule, which is why electricity bills show the energy in kWh, kilowatt-hours, units of 3600000J (3.6MJ).

The analogy generally used for distinguishing between energy and power is associated with water towers. The water present in the tower can be considered the energy (can be stored and can flow), and in turn the water flow to the exterior of the tower is the power (in particular the speed at which the stored energy flows). For a given amount of energy, it is said that can be delivered at high/low power depending on the time delay during the operation and a resource is said to be energy efficient (comparatively with another one) if uses less energy to provide the same service (Touran, 2017).

Summarizing in a very simple and clear way, we can say that energy is the total amount of work done, and power is how fast you can do it.

3.2.2 *Energetic aspects of programs*

Every processor or processor family has its own machine code instruction set. Despite the complexity of this, in practice there is a generic standard in terms of power consumption of the machine instructions. Essentially, there are three types that encompass instructions with virtually the same power consumption (Grune et al., 2012):

TYPE 1 Instructions without read / write in memory (Load_Reg R*m* ,R*n*; Add_Reg R*m* ,R*n*; Mult_Reg R*m* ,R*n*, etc.);

TYPE 2 Instructions with memory reading (Load_Mem n, R*m*; etc.);

TYPE 3 Instructions with memory writing (Store_Reg R*m* ,n; etc.);

In comparative terms, it is estimated that type 3 instructions consumed more one third than type 2, which in turn consumed one third more than type 1. In practice this means that if, for example a type 1 instruction consumes 400mA on a given machine with a fixed voltage of 1.5V, instructions of type 2 and 3 may consume approximately 530mA and 670mA, respectively. Taking these estimates into account, it is possible to draw a rather interesting conclusion that relates energy consumption and the execution time of a program. As long as the code of a program contains a balanced combination of statements of these three classes, the power consumption does not change. Therefore, the energy consumption will be directly dependent and proportional of the execution time of the program (Havinga, 2000) (Grune et al., 2012).

The maximum power dissipated during program execution is called peak power. It is an aspect with a lot of interest for the prevention of high chip temperatures, which can lead to reductions in chip lifetime, bad effect on reliability and power leakage. For extreme cases where the peak power limit is exceeded, the chip may be damaged or even destroyed. The term "peak power" is somewhat misleading because, in theory, it would correspond to the cost of the most expensive instruction in a program. However, only an instruction-length peak is short so that it has some sort of influence. Therefore, for there to be some overheating, is required a succession of these in a short but significant period of time. Thus, the time-decaying average of the power use of the instructions over a long enough time of the damage but short enough for the cooling system to be able to dissipate the heat, corresponds in practice to "peak power" (Grune et al., 2012). Some of the most common practices for lowering the average of this value are the moderate use of high-power instructions or even interspersing with lower-power instructions (Henkel and Parameswaran, 2007).

Step power is the power consumption difference between successive instructions. Controlling this aspect is also important due to the stability of power consumption it provides and also due to the influence it has on inductive noise. Large variations in power consumption can cause damage in devices and lead to reductions in battery span and a decrease in chip reliability (Henkel and Parameswaran, 2007) (Grune et al., 2012).

Step and peak power turn out to be important design constraints mainly in high-performance hardware. In the case of high-performance processors, they are even more important than average power due to the impact it has on the timing and reliability of the system (Yun

and Kim, 2001a)(Tang et al., 2001). In Figure 5 is represented a chart with the relationship between the four mentioned energetic aspects of a program execution.

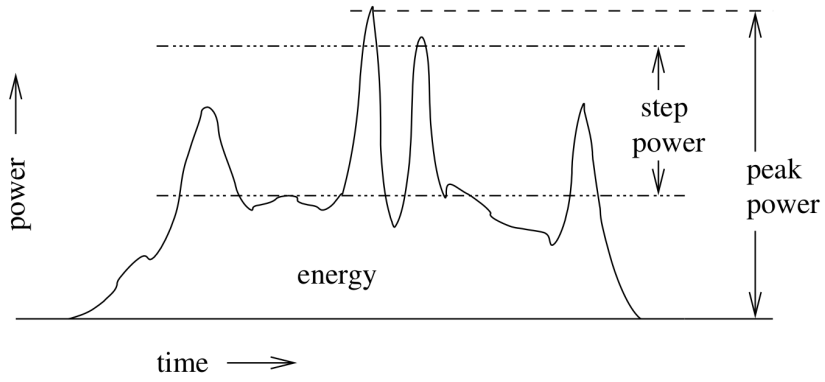


Figure 5.: Energy Consumption, Power, Step Power and Peak Power during a program execution (Grune et al., 2012).

In order to achieve a reduction on energy consumption and minimal power dissipation, several strategies are used, such as: metrics, benchmarks, energy models, etc. Within the various metrics that relate energy and power, stands out that of activity level at any given point during program execution. In this approach, peak power is the highest activity level, step power is the variation in activity levels of successive points of the program and the total amout of activities in a given program is denoted as the energy consumed (Henkel and Parameswaran, 2007).

From the point of view of Green Computing, the four energetic aspects presented have considerable importance within the different domains of action they represent. Whether it is to increase system durability, detect and prevent energy waste over time or even reduce the environmental impact from the various components, each strand allows to act and achieve considerable improvements for the intended purpose. In the particular case of the software energy efficiency during its runtime, it is verified that, along with the concerned execution time, the element of greater interest is the power consumption from the executed instructions.

3.3 OPTIMIZATION TECHNIQUES

Optimizations are considered by compile writers as a very attractive and effective way to obtain significant improvements in the generated code. Through only tunings in the program, and without neglecting its correctness or making any changes in the hardware, it

is possible to obtain improvements in several interesting aspects already addressed in this study (Pan and Eigenmann, 2006a).

Throughout this study, several possibilities have already been mentioned among distinct areas in order to reduce the energy consumption of hardware and software. The choice of algorithms and data structures faster and less memory-intensive, and the economical use of non-CPU resources such as monitors and other peripherals, are two concrete examples of the diversity and scope that these measures can have. Within the scope of compile writers, there are also optimization techniques that allow not only generate code with low power consumption but still get acceptable peak and step power properties. Although is not the main focus, and depending on the technique and the selected case study, it is still sometimes possible to obtain significant improvements in other important code aspects such as size and speed.

In this subsection we describe five optimization methods with special focus on energy issues: just compiling for speed, trading speed for power, instruction scheduling and bit switching, avoiding the dynamic scheduler and domain-specific optimizations (Grune et al., 2012).

3.3.1 *Just compiling for speed*

One of the simplest and immediate ways for the programmer to obtain gains in code energy consumption is to use some of the various existing mechanisms to improve the execution time of the same.

In chapter 4 is presented a study that demonstrates the effectiveness of this technique, in terms of CPU and memory energy consumption, using only the pre-defined optimization levels in GCC. It has been demonstrated that it is possible to make significant gains without any extra effort by the programmer who already uses the compiler to optimize the code in a generic way.

Experiments with the BURS code generator and also other researches show that the resulting codes generated to optimize the execution time, and to optimize the energy consumption are quite similar. This leads to the conclusion that, in general, code optimized for execution time is an optimum approximation of code optimized for low energy consumption (Tiwari et al., 1994) (Parikh et al., 2004).

3.3.2 *Trading speed for power*

There are several real-world applications where a response is returned before it is actually needed. Automatic processing of images that will only be used later or the generation

of reports at the end of the week that will only be visualized in the following week, are examples of recurring situations.

For cases with similar characteristics to these, a trade-off between processors speed and power can be exploited which causes, on the one hand, a reduction on the processing capacity, but, on the other hand, lower power consumption and peak power values are obtained without impairing the step power. This concept may also be applied only to the particular program areas that are considered energy critical or otherwise to low processing zones (Hsu and Kremer, 2003).

Briefly, this approach rest on the fact that most processors allow to change some of its settings voltage. In this way it is possible to benefit from a curious property which is that the power consumption is proportional to the square of the voltage, but nevertheless the speed of the processor only changes linearly with the voltage. Besides increasing the portability, autonomy and reduce costs with the power consumption of the equipment, lowering the voltage presents other benefits that make this technique quite interesting. As the voltage is lower, less heat is produced which provides better cooling of the processor. Allowing in turn an increase of its lifetime, stability and reliability; reduction of noise caused by the system; to be packaged in tighter systems; better compatibility with old applications; reduction of costs and logistics with the cooling of the system, etc. (Saputra et al., 2002) (Mueller, 2006).

Let us suppose, for the purpose of simple exemplification, that the voltage of a processor is lowered by 20%. On the one hand, this measure will have the immediate impact of reducing processing capacity by 20% and consequently the program execution time will increase by 25% (due to the need to apply a factor of 1.25 to return to the same cycle performance). But on the other hand, with a cost of 25% of processing time it is possible to obtain, among other improvements, a reduction of power consumption by 36% and this trade-off can be very advantageous for applications in which there are no really benefits of having a response in advance (Grune et al., 2012).

3.3.3 *Instruction scheduling and bit switching*

Many methods of code generation rely on the structure "instruction selection, instruction scheduling, register allocation". Optimizing the three components simultaneously turns out to be for most situations an NP-complete problem, being usually only one stage selected as a target. In particular, the phase of instruction scheduling can be replaced or adjusted according to the purpose of the generated code, namely in terms of low energy consumption. In cases where this is favorable, such as mobile and embedded devices, a number of approaches are possible through adaptations of that phase. This approach is very interesting because it can intervene in situations where other techniques do not have

space for such, because it is relatively independent of the several more common limiting factors such as chosen programming language or hardware (Grune et al., 2012).

Cold scheduling is a collection of techniques that acts precisely in the enunciated manner, exploring a particularity of electronic circuits: changing a bit to the same value (0 to 0 or 1 to 1) uses less power than switching to the inverse value (0 to 1 or 1 to 0). Combining this peculiarity with the fact that an instruction finds the bits in the processor precisely as they were left by the previous instruction, opens then a window of opportunities so that taking advantage of these factors the bits changing be minimized. This freedom has some limits because most of the bits are out of the compiler control. Yet in particular for the instruction scheduling phase the compiler has access to some bits, can inclusive rearrange instructions as desired and even later in the phase of register assignment can even choose the registers numbers with few limitations. Within the bits controlled by compiler writer, on average this collection of techniques reduces the amount of bit changes by about 30% -40% (Kandemir et al., 2002)(Grune et al., 2012).

On average, this optimization translates into a reduction of step power by 0.24 for straightforward code and a reduction of CPU power consumption roughly by 3%. Within this approach there are other techniques with better results (for example more 30% battery life), but they need some freedom to balance criteria of generated code. For example, it is possible to obtain through very aggressive instruction scheduling quite significant reductions in terms of power consumption and step power but at the expense of speed and peak power (Yun and Kim, 2001b) (Grune et al., 2012).

3.3.4 *Avoiding the dynamic scheduler*

With the advancement of technology it is possible to create smaller and smaller transistors that consume less power and produce less heat. As a consequence, it became possible to compact a larger number of transistors per area allowing to be developed new functionalities related to processing capacity, namely its speed. One of these features is dynamic scheduling being adopted by most processors today.

Commonly, the execution of instructions does not follow a predefined and absolute order but rather according to other factors such as the availability of the source operands. With dynamic scheduling it is possible for the hardware to determine an order in which instructions can be executed unlike the statically scheduled machine in which this task is left to the compiler. This technique has several advantages, such as taking into account the parallelism (which is sometimes not visible at compile time), and may not require the code to be recompiled for more efficient execution because the hardware is who deals with most of the scheduling. However, for cases where the code or parts of it are simple, the compiler

is able to set an optimal order of instructions and therefore it is preferable to turn off the dynamic scheduler in order to save energy (Grune et al., 2012).

It is estimated that about 30% of the energy consumed by the CPU is by the hardware that implements the dynamic scheduler. With only a small percentage reduction in processing capacity, it is possible to obtain a reduction in energy consumption of approximately 25% by deactivating this feature and allowing the compiler to analyze the simpler block codes (Valluri et al., 2003).

3.3.5 *Domain-specific optimizations*

Generic components and customizable for specific application domains such as embedded systems, are being increasingly used in the software development and hardware equipments. These devices are characterized by performing only a specific type of tasks such as for example Image or Video processing, PDE solving, machine learning problems and reading/processing of data obtained through a sensor. In most of these cases, there is a certain code homogeneity being very centred on certain types of algorithms or data structure (Membarth et al., 2016) (Lengauer, 2004).

The application of domain-specific optimizations techniques for the simplification of surrounding control structure, such as loop unrolling, sparse matrix representations, algebraic simplifications, multiple processing elements, dedicated logic, specialized memory and interconnection, etc., reveal themselves very interesting in terms of energy and processing speed.

There are studies that analyzes some optimization techniques related to loops and concludes that it is possible to obtain positive results, but that vary greatly according to the program and technique applied (Kandemir et al., 2000).

3.4 JUST-IN-TIME COMPILATION

In the early stages of the programming languages evolution, the compiler was responsible for producing object code (machine instructions) from a high-level language (compared to assembly), which would then be linked (by a linker) into an executable. This process can take many steps before it is optimized as machine code, but the output is always code that is ready to be executed (and that executes efficiently, as a result). C, C++, Objective-C and Go are examples of compiled programming languages widely used in IT.

At a later stage, another type of approach came to be popularized by changing the way the code is run. Compilers started compiling high-level languages to pseudo-code which would then be interpreted (by an interpreter) to run the program. This led to the removal of the necessity of object code and executables which has allowed these languages to attain

greater portability for many hardware platforms and operating systems. PHP, Python, Perl and more recently JavaScript are some examples of interpreted programming languages quite popular nowadays.

Both approaches have advantages and disadvantages when compared to each other. With the compilation it is possible to obtain a greater proximity between the code and the hardware operations performed by machine code (making it easier for programmers to control CPU usage and memory in fine detail) and also to run programs in the object language quickly and without the overhead of interpreting the source language along the way. On the other side, interpretation has as advantages the fact that interpreters are relatively easy to write and the possibility of monitor what a program tries to do as it runs (to enforce a policy, say, for security).

Just-In-Time (JIT) compilers, also known as dynamic translators, are the next-generation of compilers; compile-time and execution-time are not any more completely separated phases (as they were in the traditional approaches), they are merged and the compiler generates code by need, executes it and also monitorize the execution optimizing the process on demand. A JIT compiler runs after the program has started and compiles the code (usually bytecode or some kind of VM instructions) on the fly (on demand, when it's needed) into a form that is usually faster, typically the host CPU's native instruction set. So instead of interpreting bytecode every time a method is invoked, will compile the bytecode into the machine code instructions of the running machine, and then invoke this object code instead. This approach has access to dynamic runtime information whereas a standard compiler doesn't and so can make better optimizations like inlining functions that are used frequently or analyze the flow of control inside a method (or specific sections of it) and rearrange code paths to improve their efficiency. With this approach it is possible to obtain the best of both worlds: reduce the CPU's workload by not compiling everything at once, obtain an optimized execution for that particular CPU and keep portability running on any operating system or hardware platform (Wodehouse, 2017) (Grune et al., 2012).

JIT compilers are already part of some notable frameworks such as Java Virtual Machine[1], Common Language Runtime[2], Dalvik Virtual Machine[3] or Android RunTime[4] (in newer versions), Zend Engine[5], crto[6] and Node.js[7].

Ideally, the efficiency of running object code will overcome the inefficiency of recompiling the program every time it runs. If the overhead inherent to the compilation process is

1 https://docs.oracle.com/javase/specs/jvms/se8/html/index.html
2 https://docs.microsoft.com/en-us/dotnet/standard/clr
3 https://source.android.com/devices/tech/dalvik/
4 https://source.android.com/devices/tech/dalvik/
5 http://www.zend.com/en/products/zend_server#engine
6 https://www.embecosm.com/appnotes/ean9/html/ch05s02.html
7 https://nodejs.org/en/

too high, this technique is no longer acceptable. Modern JIT compilers need to be well tuned and implement advanced techniques in order to achieve their objectives. Select very carefully and compile only ”hot spots” of the program, preloading on embedded systems of code extremely optimized for a specific language or even use multiple compilations quality levels- fast for most code and more sophisticated for critical areas — are some examples of already adopted techniques (Jung et al., 2011)(Grune et al., 2012).

At the energy level there are also benefits in this promising approach. JIT optimizations can greatly optimize the code, namely to reduce significantly the instruction counts which is directly reflected either in a higher energy efficiency or in a decrease of average power dissipated by a program (Lane and John, 2006).

3.5 SUMMARY

Although low energy consumption is not yet considered a priority in the compilation process, it is nevertheless already considered as an optimal code generation property along with correctness (of the translation scheme), high speed (concerning the target program execution) and small size (in terms of the target code length, or stage memory needs).

In this chapter, some of the techniques most used used in different scopes for code optimization concerning the energy consumption were described. In a generic way they can be ordered by energy efficiency as follows:

1. Making the execution of the program faster has direct effects on the energy consumption of the program;
2. Lowering the CPU voltage reduces the speed of the program linearly but also the power consumed quadratically;
3. Reduce the amount of bit switching through tuning the instruction scheduling and use low-energy instructions (cold scheduling) as well as avoid access to memory when possible.
4. Specific domain-dependent methods appear as a good alternative especially for specific hardware applications such as embedded systems.

In addition to the optimization techniques for code generation, there are other more comprehensive approaches that also explore the achievement of significant energy gains. Just-In-Time compilation is one of such initiative that tries to take advantage of compiling and interpreting. By compiling the code in the instant before it is executed, it can boost various advantages such as speed, increased portability, energy efficiency, among others.

Over the years, compilers have made considerable efforts to keep pace with the impressive evolution of hardware. Transistors increasingly reduced in size and present in larger numbers per component, parallel execution units, multiple cache levels, etc. has forced the development of increasingly sophisticated optimization techniques by compilers, under penalty of becoming old-fashioned.

The constant increase in hardware complexity has been curiously one of the biggest factors in ineffectiveness of compiler optimizations. As it is more and more difficult for compilers to predict the actual effect of an optimization, hardware started to be built with the self ability to optimize the code. Various techniques implemented in hardware, such as dynamic scheduling, reflect optimizations traditionally performed by compilers. Ironically, these more sophisticated implementations make it even harder to keep up with the race on the part of the compilers. Complex hardware architectures also require better compiler optimizations because otherwise it is not possible to fully enjoy the features offered achieving significant increase in energy consumption.

With the increased pressure on the use of parallel processing capabilities it is also necessary to improve the programming languages. The most used are based on the sequential execution model of a single instruction stream and this disparity with modern processors, among other reasons, makes generating efficient code for each type of processor far from a trivial task. Since this is clearly not one of the concerns of compilers, it remains for the programmer the efficient use of parallelism through mechanisms such as multiple parallel instruction streams (Grune et al., 2012).

In addition to what has been observed throughout this chapter, there are also other approaches that apply more intensively green strategies in order to produce energy conservative executables.

Green Compilers are a good example of such tools, proving to be especially important in more demanding situations such as embedded and large scale systems. Among the various techniques they use, the following stands out: cache skipping, use of register operands, instruction clustering, instruction re-ordering and memory addressing, use of energy cost database, loop optimization, dynamic power management, resource hibernation, cloud aware task mapping and eliminate recursion (Fakhar et al., 2012) (Fakhar et al., 2011) (Chanda et al., 2017).

Currently there are already on the market several options that follow this methodology of code generation. In particular, Green Hill compiler[8] and ENCC[9] apply some of the techniques described for programs developed in C++. However there is still some scarcity in terms of quantity of products, variety of paradigms that consider and how they are made available (mainly of free or open source options). Due to these factors and the dis-

8 https://www.ghs.com/products/compiler.html
9 https://ls12-www.cs.tu-dortmund.de/daes/forschung/energy-aware-c-compiler/download.html

advantages inherent in this kind of tools (notably in the trade-off between performance and energy consumption), unfortunately they are still not widely used by the community. Nevertheless, with the increasing awareness and dissemination of Green Computing topics, it is possible to predict that this situation may change in the near future. As soon more diversified and sophisticated products emerge, certainly will also increase the adoption of this type of technology by the developers.

4

IMPACT OF GCC OPTIMIZATION LEVELS IN ENERGY CONSUMPTION DURING PROGRAM EXECUTION

In this chapter, we address the impact that running optimized compiled code has in the CPU energy consumption. In particular, the study is focused on programs compiled by GCC with optimization suits for a target machine based on an Intel CPU. Although the CPU is the most important computational resource of this study it was also examined the impact in memory and GPU in order to perform a more complete analysis.

The chosen programming languages are C and C++ according to the general decision taken in the context of GreenSSCM. In addition, they are also nowadays two of the programming languages more used (TIOBE Index).

For C/C++ compiler was chosen GCC[1](v.5.3.0) because it is a robust option, well documented, the most widely used and this allows different code optimization levels.

Taking into consideration that GCC is an integrated distribution of compilers for several major programming languages (GCC team, 2016c), two more programming languages have been added to this study. In this way we could analyze in more detail the behavior of GCC and its optimization procedures to get a deeper knowledge about the impact of optimizations on energy consumption for a larger and more diverse set of programming environments. After a preliminary analysis, we decided to consider Objective-C and Go because they are also widely used today (TIOBE Index) and still subject of improvements in the most recent GCC versions (unlike Java, for example, for which there is not any relevant update since version 4.5 of April 2010 (GCC team, 2016b) (GCC team, 2016a)). Although Apple has decided to completely replace the use of GCC by LLVM[2] and Clang[3] in their systems to compile Objective-C programs, actually Objective-C was not "forgotten" by GCC and, albeit some stagnation in support and improvement, it makes sense to study the impact of GCC compilation on Objective-C runtime performance (Dilger, 2016).

1 https://gcc.gnu.org/
2 http://llvm.org/
3 http://clang.llvm.org

After choosing the source languages to compile with GCC, the last fundamental decision to complete the experiment setup is concerned with the choice of the target machine. The assembly code that will be generated, and the optimization effects/impacts that can be reached, depend, of course, on the machine selected. Accounting for that Intel is the manufacturer with the largest market share in recent years (57% in 2012, 65% in 2013), and is currently present in over 80 percent of the computers sold worldwide, it is imperative to use an Intel microprocessor for more relevant results (more information in 2.3) (ITCandor, 2012)(ITCandor, 2013) (King, 2015).

The elements gathered for this study, as well as all the results obtained, are also presented in the project website[4].

This chapter is organized as follows. In Section 4.1 it is referred a similar study in our context for embedded systems. In Section 4.2 the main elements are described to carry out this experimental study. In Section 4.3, the GCC optimizations are specified and explained the measurement process. In Section 4.4 are shown and analyzed the results obtained. Lastly, in Section 4.5 a conclusion is devised focusing the main considerations of the present study.

4.1 RELATED WORK

Despite the fact that energy consumption subject is only a research topic in recent years (largely due to the massive widespread of quite advanced wireless and mobile devices), already at the beginning of this century appeared some projects considering compilers precisely as a way to combat it. In 2001 the standard optimization levels -O1 to -O4 were evaluated to understand the effect of a few individual optimization of DEC Alpha's cc compiler on power and energy consumption of the processor. The authors concluded that when the optimizations decrement the number of instructions to be executed, also the energy consumption is reduced (Valluri and John, 2001). In the same year, another study was conducted to explore the effect of the compilers for existing processor architectures addressing the same problem. They concluded that the compiler optimizations has enough potential to achieve some reduction in energy consumption, but it would be necessary to expose more innovating micro-architectural features to the compilers, in order to obtain substantial gains in energy saving (Chakrapani et al., 2001).

There are also some studies in which algorithms are designed to select combinations of compiler optimization flags that, for a given input program, generate a machine code with a better performance at runtime (without taking into account the energy issues or flags that

4 www.di.uminho.pt/~gepl/OCGREC/projects/project1.html

have emerged in more recent versions of GCC) (Pan and Eigenmann, 2006b)(Patyk et al., 2009)(Pallister et al., 2013). If there is indeed a relationship between the optimizations applied by the GCC and the energy consumption of programs, such algorithms (or variants thereof) may be very useful. This investigation will be left for future work.

Considerations for energy efficiency are especially relevant at the level of embedded platforms. In (Pallister et al., 2013) they use GCC, 10 benchmarks and 5 different embedded platforms to analyze the energy consumption of a large number of compile options. Through hardware power measurements and some case studies explore various hypotheses and conclude, among other things, the execution time and energy consumption are correlated in most general cases.

Until this day there is very little work that explores widely the impact that the various compilation options provided by compilers (including GCC) has on the energy consumption of the software that use them (Pallister et al., 2013).

4.2 EXPERIMENTAL SETUP

The three main elements to carry out this experimental study are: a platform for taking measurements (a laptop); the software that makes measurements; and also the software packages that will be measured.

4.2.1 *Testing Platform*

The study was accomplished on a laptop Asus N56JN-DM127H, running under Linux. The hardware/software resources most relevant characteristics for the required analysis are: Arch Linux 64-bit (Linux Kernel 4.4.5-1); Intel® Core i7-4710HQ up to 3.5 GHz, Haswell Family; 8 GB DDR3L 1600MHz; and NVIDIA® GeForce® GT 840M, 2GB DDR3 VRAM.

4.2.2 *Measurement Software*

The energy measurement necessary to make the comparative study was performed using Running Average Power Limit[5](RAPL) interface. RAPL allows, among other features to read the Machine-Specific Registers containing information about the energy consumed by the CPU, RAM and GPU during a given period of time (Intel Corporation, 2015)(Hähnel et al., 2012).

5 https://01.org/blogs/tlcounts/2014/running-average-power-limit-—rapl

Performance Application Programming Interface[6](PAPI) and the Perf[7] were also considered as alternatives to RAPL to take the desired measures. However after analyzing pros and cons of each one, these options were discarded. On one hand PAPI is an event-driven tool what makes its use inadequate in our context. On the other hand, Perf only allows to obtain information about the CPU and we were interested in investigating the other possible sources of consumption, the memory and the GPU.

RAPL was used through an extension[8] developed by the team in which context this study was undertaken; that tool (RAPL extension) simply reads that register. This extension keeps all other features previously present, such as setting the number of the CPU cores that will be measured, and adds the ability to measure the consumption of a certain operation and also the time spent for its completion. For this study the operation above referred is the compilation and/or the execution of a program, being only necessary to inform the path to the respective makefile or the executable. This extension can also be used in other areas of study, because it allows to measure the compilation of a program in any language that contains their dependencies expressed in a makefile or even any other executable. The distinct features that this tool provides are managed through a mechanism of flags passed as arguments when the tool is invoked.

4.2.3 *Measured Software*

In the experiment described in this chapter we have analyzed 12 programs that differ in some aspects such as: programming language, main objective, code complexity, external dependencies and also compile and execution time. Despite these differences, were all chosen according to the following criteria:

- Open source code, allowing to analyze in more detail the complexity and how the objectives are achieved;
- Running under Linux environment;
- Coded in one of the programming languages chosen in the context of GreenSSCM project, C or C++, Objective-C or Go;
- Not interactive, this is independent of user interaction during execution, thus avoiding waiting for input that would have interference in the measured values and also allows automate this part of the process;

6 http://icl.cs.utk.edu/papi/
7 https://perf.wiki.kernel.org/index.php/Main_Page
8 https://github.com/deater/uarch-configure/tree/master/rapl-read

- No graphical interface;
- Total execution time less than 60s, to prevent possible overflows.

Although the many particularities of the chosen programs, there are some characteristics common to all of them that can be used in order to be possible to perform a comparative study, without knowing in detail the code of each studied programs.

In Table 1 we characterize the selected programs regarding some of their features. The parameters considered are: (PL)programming language; the (IF)input and (OF)output files; (CC)code complexity (low, medium or high); and the (ED)amount of External Dependencies (none, few, several or lot). Average compile and execution times(ACT, AET) are also included just to figure out a quantitative description of their size/complexity.

Program	**PL**	**IF**	**OF**	**CC**	**ED**	**ACT (s)**	**AET (s)**
MMC	C	None	None	Low	None	0.17	26.16
Grades	C (Flex and Yacc)	Txt	Html	Low	Few	0.23	32.84
Bzip	C	Wav	Bz2	Medium	None	1.47	23.41
Bzip2	C	Wav	Bz2	Medium	Several	1.97	23.38
Oggenc	C	Wav	Ogg	High	None	3.83	22.91
Pbrt	C++	Pbrt	Exr	High	Lot	43.08	19.42
Matmul	Go	None	None	Low	None	0.13	15.05
PGo	Go	None	None	Low	None	0.15	13.06
Sudoku	Go	Txt	None	Medium	None	0.24	24.43
Matmulobjc	Obj-C	None	None	Low	None	0.19	3.88
Miscellany	Obj-C	None	None	Medium	None	0.40	9.47
Sorting	Obj-C	None	None	Medium	Few	0.26	35.62

Table 1.: Some features of the measured programs.

A brief description of each one of the subject programs follows:

MMC: Multiplication of matrices with size 1024x1024 using 6 different methods.

GRADES: Generates the final grades of the students in a course from their marks.

BZIP AND BZIP2: File compression tools (replacing the input file).

OGGENC: Perform file format conversion (creating a new file).

PBRT: Executes the rendering of images using ray tracing.

MATMUL: Multiplication of two matrices with size 2000x2000.

PGO: Modular random level generator (roguelike type).

SUDOKU: Solves 20 extremely hard Sudokus repeated 1024 times;

MATMULOBJC: Multiplication of matrices with size 800x800 using 6 different methods.

MISCELLANY: Collection (more than 1000 lines) of practical exercises.

SORTING: Applies 6 of the best known sorting algorithm to an array of 8000 positions.

4.3 METHODOLOGY

After defining the software and hardware environment for the study, and the set of programs to test, in this section we will describe the main decisions taken in what concerns the parameterization of GCC in order to configure it for fair comparisons. We also define the strategies followed to get the measures and to obtain significant statistical results.

4.3.1 *Optimizations Flags*

The GCC compiler has hundreds of flags related to the optimization of the generated code specific to a given machine. Due to the specificity of each of the flags and the fact that sometimes they are mutually exclusive, this study will only focus on the several optimization levels specified by the compiler switch -O. There are 7 different levels of optimization and each contains a number of individual flags that are enabled or disabled (depending of the level). Below are described the referred levels with some more detail (GCC team, 2014)(Saddler, 2016).

-O0: Default level (disables all optimization flags). Reduce compilation time and make debugging produce the expected results.

-O1: Basic optimization level (enables up to 39 flags). Reduce code size and execution time without taking much compilation time.

-O2: Recommended optimization level (enables up to 73 flags). Enables all existing in -O1 and the remaining that do not include a space-speed trade-off, increasing both compilation time and the performance of the generated code.

-Os: Reduced code size (enables up to 65 flags). Enables all -O2 options that do not increase the size of the generated code. Useful for target machines with limited disk storage space and/or CPUs with small cache sizes (with dramatic improvement of performance).

-O3: Highest level of optimization (enables up to 82 flags). Enables all existing in -O2 and further some very heavy in both time compile and memory usage terms. It is not guaranteed that the code generated is better in terms of performance than the previous set.

-Ofast: Disregard strict standards compliance optimization (enables up to 85 flags). Enables all -O3 options and more 3 that are not valid for all standard-compliant programs.

-Og: Optimize debugging experience (enables up to 74 flags). Offer a good debugging experience enabling all optimizations that do not interfere with debugging and reasonable level of optimization while maintaining fast compilation.

After a preliminary test phase, we decided not taking into consideration the level -Og for this study because this level performs optimizations which do not contribute to an energy reduction compared to the other 6 discussed.

4.3.2 *Measurement Process*

After having selected the programs to be studied and defined the set of optimizations to be compared, it was mandatory to setup the measurement process to apply to all programs in order to cover all the desired cases. This process can be described by the following steps:

1. Choose a program and its Makefile;
2. Choose the desired optimization level;
3. Execute 100 times the measuring tool for the program and the chosen optimization level;
4. Process the output generated by each invocation of the measuring tool:
   a) Get the energy consumption and time values;
   b) Ignore the 10 highest and lowest values;
   c) Compute the average of the remaining 80 values;
   d) Generate a table and plot with the results in an HTML page.
5. Repeat step 2, 3 and 4 for all 6 levels of optimization;

6. Repeat step 1 to 5 for all 12 programs.

All measurements relating to a program were performed uninterruptedly while avoiding fluctuations related to the testing platform (e.g. pre-loading of data, memory heating, etc.). During the measurement process all executions of the intended programs were forced to run on just one core of the CPU, through the tool flag -n, to ignore efficiency issues related to the parallelization that some programs may allow. The processing of the output (referred in item 4 above) was done using essentially PERL[9](for parsing the results of each operation) and GNUPlot[10](to generate plots of each program).

4.4 DISCUSSION OF RESULTS

To illustrate the results obtained, five examples of charts are displayed: three running C/C++ (Figure 6, Figure 7 and Figure 8), one running Go (Figure 9) and one more example running Objective-C (Figure 10).

The remaining charts, as well as the HTML pages, can be found in the online repository[11] and website[12] of this project.

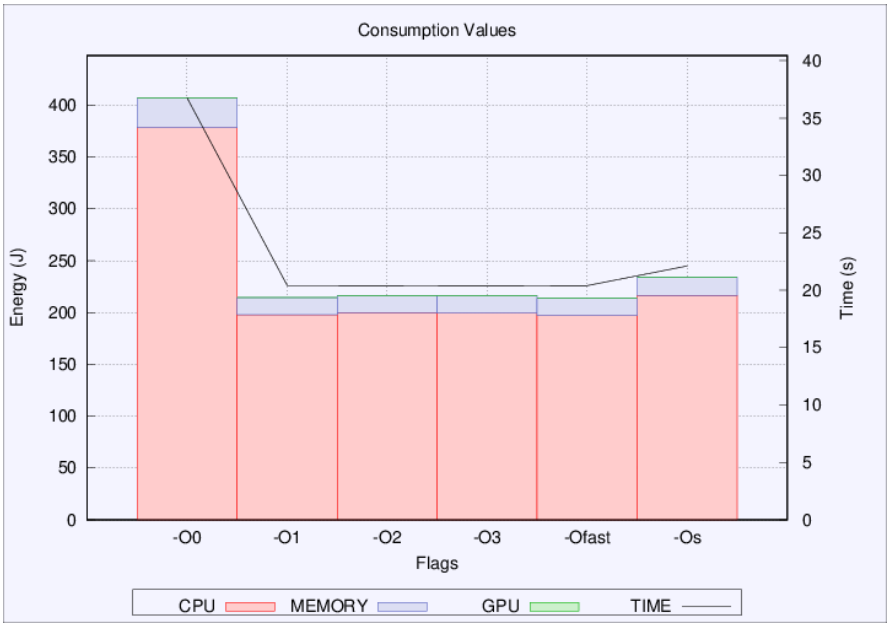


Figure 6.: Results of Bzip measurements (C program).

9 https://www.perl.org/
10 http://www.gnuplot.info/
11 https://github.com/david-branco/gcc-optimization-energy-article-extended
12 www.di.uminho.pt/~gepl/OCGREC/projects/project1.html

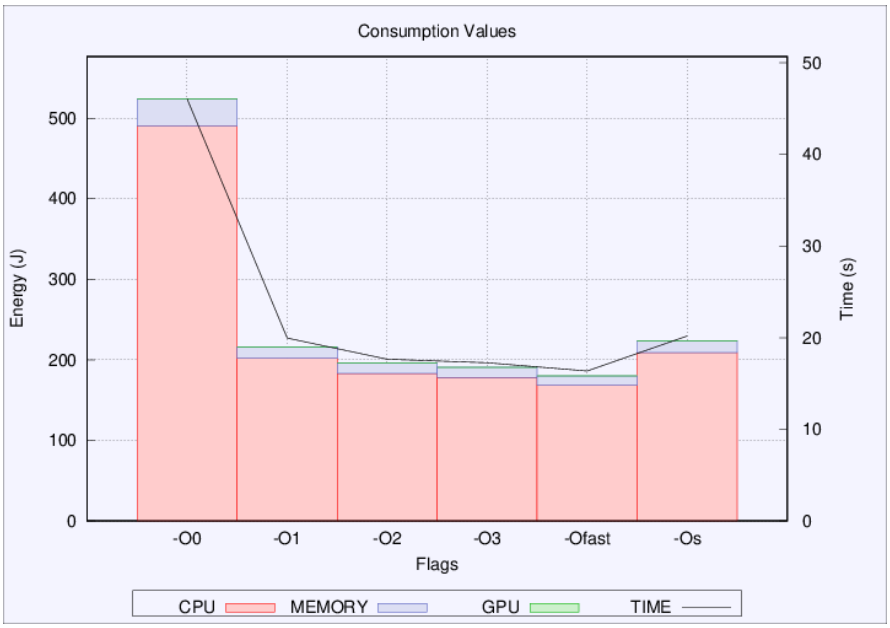


Figure 7.: Results of Oggenc measurements (C program).

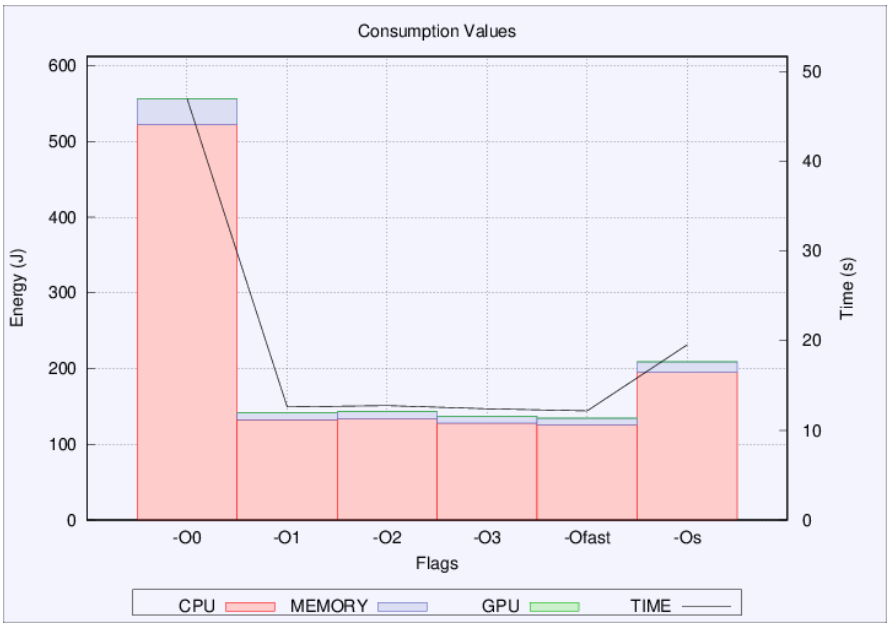


Figure 8.: Results of Pbrt measurements (C++ program).

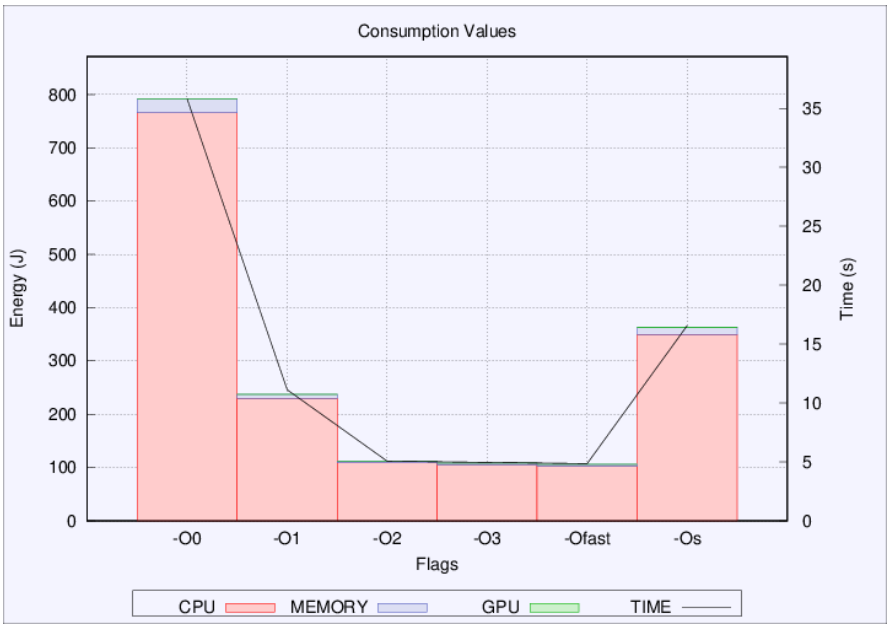


Figure 9.: Results of PGo measurements (Go program).

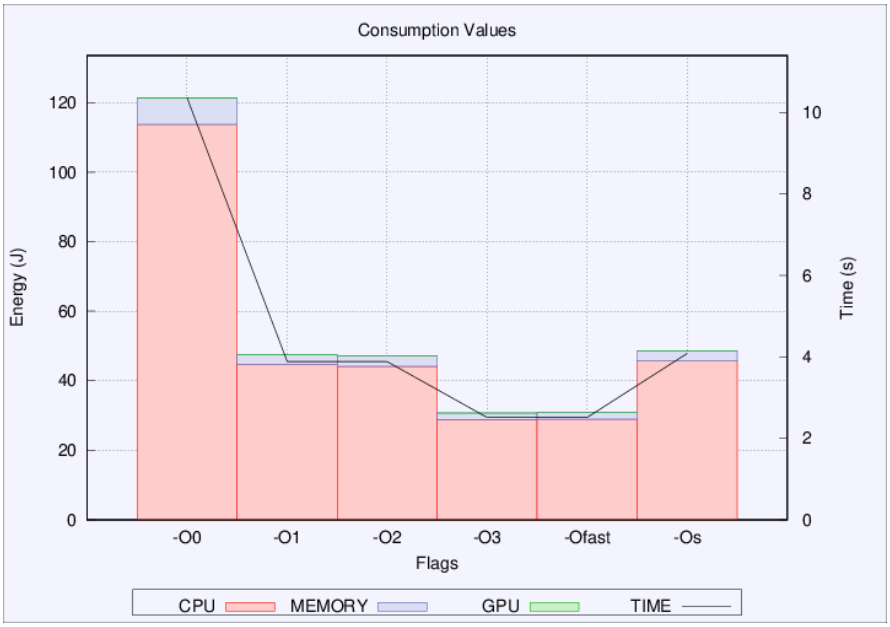


Figure 10.: Results of Matmulobjc measurements (Objective-C program).

In all analyzed charts it was found that, due to the classes of the programs selected, the energy consumed by the GPU is reduced. It is clear that energy and time consumptions are undoubtedly lower when selected an optimization level different from the default, and that the optimization is greater for more complex source codes (in the case of *Pbrt* there was a

reduction of almost 76%). For low complexity programs it was confirmed that none of these levels is much higher or lower than the remaining, with only a minimal difference (in the extreme case may not even exist) among some of them because of individual optimizations that each enables/disables.

Program \ Level	-O0	-O1	-O2	-O3	-Ofast	-Os
MMC	34.361	24.552	24.402	23.649	24.265	25.740
Grades	40.038	32.378	31.239	31.226	31.060	31.091
Bzip	36.755	20.408	20.396	20.361	20.399	22.115
Bzip2	36.755	20.493	20.366	20.477	20.427	21.773
Oggenc	46.068	19.958	17.648	17.256	16.359	20.177
Pbrt	47.008	12.614	12.761	12.413	12.175	19.547

Table 2.: Execution times of C/C++ programs in seconds by optimization level.

Program \ Level	-O0	-O1	-O2	-O3	-Ofast	-Os
MMC	354.934	237.568	239.599	240.192	238.445	227.809
Grades	352.228	257.912	246.786	248.958	245.701	249.664
Bzip	406.914	214.438	216.539	215.882	213.496	234.162
Bzip2	402.516	212.324	212.327	215.639	213.947	229.156
Oggenc	523.992	216.264	195.844	190.472	180.038	213.51
Pbrt	556.222	141.461	142.815	136.84	134.378	209.164

Table 3.: Selected C/C++ programs and their energy consumption (CPU and memory) in Joules by optimization level.

Considering all selected programs and optimization levels which are not the default, analyzing the execution time (Tables 2, 4 and 6), memory and CPU energy consumption (Tables 3, 5 and 7), and ignoring minimal differences of values between levels, it appears that in most cases the -Ofast level is the most efficient unlike -O1 and -Os options. It is also perceptible that, although there is no much difference between the -O2 and -O3 levels, -O3 is slightly more efficient especially when the program complexity increases. Although regarding the results presented, -Os was not one of the best levels in the analyzed

parameters, it is know that -Os has potential for significant improvement in some particular cases, for example where their optimizations are capable to fit the code in the cache.

Level / Program	-O0	-O1	-O2	-O3	-Ofast	-Os
Matmul	22.247	21.396	8.420	8.429	8.434	21.347
PGo	35.803	11.117	5.066	4.961	4.853	16.567
Sudoku	36.586	24.557	14.951	13.156	13.162	26.169

Table 4.: Execution times of Go programs in seconds by optimization level.

Level / Program	-O0	-O1	-O2	-O3	-Ofast	-Os
Matmul	290.775	246.092	105.301	105.514	105.541	231.753
PGo	791.741	237.263	112.529	108.662	106.343	362.635
Sudoku	414.359	265.79	164.581	145.177	146.065	286.753

Table 5.: Selected Go programs and their energy consumption (CPU and memory) in Joules by optimization level.

Level / Program	-O0	-O1	-O2	-O3	-Ofast	-Os
Matmulobjc	10.354	3.885	3.882	2.511	2.513	4.085
Miscellany	22.442	13.681	9.038	10.447	10.448	13.665
Sorting	37.623	35.692	35.663	35.787	35.748	36.031

Table 6.: Execution times of Objective-C programs in seconds by optimization level.

One of the main objectives of this work was to determine if programs optimized at compilation time also have an optimized energy consumption during execution. Although the data obtained clearly demonstrate a great optimization of energy consumption, when selected optimization levels which are not the default, however it is also noticeable that the execution time of programs previously optimized also decreases dramatically (generally for shorter times was obtained lower consumption).

Program \ Level	-O0	-O1	-O2	-O3	-Ofast	-Os
Matmulobjc	121.262	47.532	47.053	30.714	30.867	48.664
Miscellany	248.844	129.711	95.549	107.859	107.713	129.914
Sorting	456.098	435.695	435.908	439.173	433.084	438.992

Table 7.: Selected Objective-C programs and their energy consumption (CPU and memory) in Joules by optimization level.

Thus, it is not possible to conclude with certainty GCC's strategies on the matter. We saw that in all cases energy consumption is directly related to execution time. In fact analyzing all graphs, that depict the data collected along the experiment, we can say that the timeline follows the trend of the columns with the consumption of each component by optimization flag.

4.5 CONCLUSION

In this chapter we described an experimental test aimed at studying the impact of GCC optimization on the energy consumed by the compiled C, C++, Go and Objective-C programs at runtime. We concluded that energy decreases as faster is the code and so we can affirm that GCC optimizations techniques have a positive impact in favor of green computing concerns. To software developers, this conclusion means that they do not need an extra effort if they decide to have their code more efficient concerning both execution time and energy consumption.

The framework developed to perform the necessary measurements is also an important result of this work. It was developed in a rather generic and comprehensive manner, allowing the analysis of several operations and programming languages, in order to be possible its usage in other GreenSSCM projects. Also, the overall output produced by the measurements performed in this study is important because it can be used as a good workbench for other green oriented research.

The obtained results are in line with what was the initial intuition of GreenSSCM team members and also the conclusions already reported in (Pallister et al., 2013) for embedded systems. After finishing this experimental work, we were aware that the time-energy relationship was already described and explained in the Technical Report (Choi et al., 2013),

corroborating our experimental findings.

However a lot of work remains to be done to understand the strategies followed in those optimization algorithms in order to understand if there is still room for more reduction. The study of the optimizations impact on programs that use parallelism, the analysis of more programming languages that GCC can handle or investigate programs that run on GPU are some of the possible research directions aiming at directly complement the work here reported.

Concluded the study and taking into account the results obtained, namely the fact that there are some improvements in energy consumption produced by compiler optimizations (albeit indirectly), some other working hypotheses rise up to proceed within this scope. A first one is to study the application of the algorithms referred in Section 4.1 in order to obtain specific sets of flags that can further reduce energy consumption. Another one is to look for special types of machine instructions that can be chosen by the compiler during the code generation/optimization phase regarding the energy consumption. This is, we intend to analyze the information provided by the current machines' Instruction Sets to verify if the energy consumption cost is provided (explicitly available) to be considered by the compiler's optimization algorithms.

5

IMPACT OF COMPILATION BY INTEGRATED DEVELOPMENT ENVIRONMENTS IN ENERGY CONSUMPTION DURING PROGRAM EXECUTION

Since its inception programming tools have proved to be essential tools for any type of programmer. Be it a student who is taking his first steps in the field or a very experienced and multifaceted developer, all resort to mechanisms that make programming work more effective. They are used every day for the most diverse tasks, whether they are to create, edit, debug, maintain and/or perform any programming or development-specific task. They are available in the most diverse formats, from a simple source code editor and a compiler or interpreter, to a work environment with graphical interface and quite advanced features.

Within the various existing formats, Integrated Development Environments (IDEs) are clearly the most successful tools. They are able to provide in a single workspace a set of very advantageous features that allow the developer to increase his productivity. Among the most common features are: sophisticated source code editors, graphical user interface, build automation tools, debuggers, version control systems, hierarchy diagrams, etc. Due to the inherent advantages of its functionalities, this type of development tools has gained a lot of acceptance over the last 30 years by the community which has led to the creation of thousands of solutions.

An interesting feature of IDEs are the compilation profiles that allow automatically get an executable according to a set of predefined parameters. Given this particularity and the research done in the context of the present masters' work, the opportunity arises to apply the knowledge acquired to such an important family of tools.

In this chapter, an experimental study is carried out, which intends to investigate from an energy perspective the performance of executables generated by the IDEs. They will be analyzed mainly from the perspective of their execution time, CPU and memory RAM energy consumption (individually and together) and the ratio between both factors (Energy/Time). This subject allows to continue to explore in more detail the role of the compiler and its compilation parameters in the considered strands. There is also room for other subjects of

analysis, namely how to compare the tools according to these results, what kind of options they provide to users, what is the performance of the hardware and its components, among many others.

Initially, it was necessary to investigate in detail some of the most relevant aspects of the IDEs and how they could be used according to the intended purpose. In particular, what types of compilation profiles exist, how they differ or how their content is extracted. It was also necessary to study the market of this type of tools and to investigate the most important options and formats plus new trends that might be of interest for analysis.

Next it was necessary to define in concrete which elements would be part of the remaining study. Taking advantage of the knowledge acquired previously, some principles and methodologies were applied as well as the reuse of some parts which stood out positively and which still remained relevant. The choice of the compiler was once again GCC (7.2.0) which, in addition to all the previously mentioned aspects, is compatible with the type of tools to be analyzed.

After observing that the programming languages supported by GCC have a similar tendency for all analyzed cases, in this chapter we have chosen to select only one option as a study target. In this way it is possible to analyze in more detail some peculiarities of the language and of the compiler itself. The selected language was C that in addition to the reasons presented previously, displayed excellent results in the previous study and has a strong integration with both the selected compiler and the type of tools intended.

After selecting these two elements, it was essential to study in depth the parameters used in the compilation process. In a first phase analyze the compiler and the categorization that it provides and later fit the options present in the compilation profiles obtained for the language in question.

Within the reused elements of the previous study, stand out the target machine and the measurement framework.

The target machine has not undergone any changes to its hardware, which possesses a modern microprocessor from Intel (the leading manufacturer of the market). However, an update was made on most of the software it owned, namely the Linux kernel and Operating System.

One of the versatilities of the measurement framework conceived in the previous study is that it had been developed quite generically allowing it to be used in other areas of analysis as well. Demonstrating precisely this, the tool is again used to carry out the necessary measurements. Among the advantages listed it presents very relevant virtues taking into

account the hardware used and fits perfectly into the intended analysis.

One of the points of analysis which was more detailed than the previous study was the set of benchmarks used. A set that would be more challenging for the hardware used was intended, one that would have more diversified scopes and backgrounds and still relevant for the community in general. The choice fell on the Computer Language Benchmarks Game[1] project (CLBG) which in addition to these factors has many other interesting features for the desired analysis.

The elements gathered for this study, as well as all the results obtained, are also presented in the project website[2].

This chapter is organized as follows. Initially in Section 5.1 some relevant aspects about this type of tools are analyzed. Then, Section 5.2 introduces the most interesting aspects of the types of options that the tools provide. In Section 5.3 is presented an interesting project (CLBG) that has software solutions for various programming challenges in the most diverse languages. In Section 5.4 three of the main elements of the study are described: testing platform, the measurement software and the measured software. In Section 5.5 the studied tools are presented and is performed the description of their compilation profiles and parameters. Still within the same section, the approach adopted in the measurement process is explained. Afterwards, the results obtained in the experimental study are presented and discussed in Section 5.6. Finally, the Section 5.7 addresses some of the considerations of the presented study.

5.1 INTEGRATED DEVELOPMENT ENVIRONMENTS

Following the technological advance of the last 50 years, the developers work methodology was also progressing in order to maximize their productivity and take better advantage of the new capacities that were provided. Gone are the days when a developer needed to write the code in a text editor, save it, exit the editor, run the compiler, annotate the error messages in an auxiliary pad, and finally re-inspect the code. Some methods and practices were developed in order to streamline the whole process and one of the early highlights was the creation of a software suite called Integrated Development Environment (Patrizio, 2013).

The first time an integrated editor and compiler emerged was with the release of Turbo Pascal[3] in 1983, after Borland Ltd purchasing a Pascal compiler. In addition to other ad-

1 https://benchmarksgame-team.pages.debian.net/benchmarksgame/
2 www.di.uminho.pt/~gepl/OCGREC/projects/project2.html
3 http://turbopascal.org/

vantages such as ability to run in memory, developers could from there have the error messages in the text editor itself and with a click go to their precise location in the source code. Thanks to the advancement in graphical computer aspects and following the ideas launched by Turbo Pascal, in 1991 appears which is considered by many the first true IDE, the Microsoft's Visual Basic[4]. Built in one of the most popular languages of the 80, it quickly gained notoriety due to the great increase of productivity since it allowed to think about programming in graphical terms and not just in the textual context (Veracode, 2018) (Patrizio, 2013).

Nowadays they are already deeply rooted in software development and the number of existing options are quite outstanding. From various price ranges, communities from which they emerge or even different technologies and target audience, all have different capabilities that make them so sought after by developers and software companies.

In this section are presented some concepts about this method of software development so in vogue these days. Some advantages and disadvantages of using IDEs are identified and which differentiating factors exist between them. In the sequel a study is presented that deepens some of the information presented here as well as explored new aspects such as what types of executables they produce.

5.1.1 *Meaning and Main Features*

An Integrated Development Environment is a software application that in a single work environment provides development tools in an integrated way to increase programmer productivity. They are designed to agglomerate all the basic tools inherent to the different stages of software development, as well as some over-the-top functionality, providing a programming environment to streamline developing. Among the tools often available, stand out for their importance and usefulness (Veracode, 2018):

SOURCE CODE EDITOR

It is a text editor specifically designed for the writing/editing of computer programs source code. They are, therefore, a fundamental tool for programmers and distinguish themselves from text editors because it has the ability to simplify and improve the writing/editing of source code.

COMPILER

Essential tool that intervene as mediators between programmers and machines in the developing code process (more information in Chapter 3). Some IDEs even allow to

4 https://docs.microsoft.com/en-us/dotnet/visual-basic/

select which manufacturer or version you want to use as well as choosing different compile/interpretation profiles and parameters.

DEBUGGER

This tool is widely used by programmers to search for development errors and to test application programs. In addition to a minimalist graphical interface simulating the multiple steps of a program execution, some IDEs still provide several operations modes (e.g. full or partial) to limit the impact of the resulting code slower speed.

BUILD AUTOMATION TOOLS

Build Automation Tools (BATs) allow to automate simple and repetitive but essential tasks for the development of software such us compiling, packaging and testing. Integrated in IDEs they allow with a click to build the entire project in a very comfortable, fast, consistent and secure way, reducing the space for errors and other complications inherent to the execution of the several steps. Another great advantage is the simplified management of the target environment and dependencies on third party software, being possible to validate and define in particular which requirements are necessary for the project (inclusively specify the version) and even to install them.

These functionalities are so important that sometimes IDEs do not develop them, choosing instead to integrate, trust and delegate the task to external tools such as GNU Make[5], Maven[6], Ant[7] and Gradle[8]. Typically, predefined sets of configurations are provided and it is possible through commands and file manipulation using a scripting language to define the intended process. The use of these tools is seen as a good practice in software development and a step in moving toward a continuous delivery model and a better relationship between Development and IT Operations.

EXTRA FEATURES

Despite the more generic tools that most IDEs have, there are also some extra features that, depending on the cases, are also very relevant, such as: intelligent code completion and expansion of abbreviations; bracket and code highlighter; automatic import of libraries; code linting and error diagnostics; useful information on sidebars such us hierarchy diagram, project browser and multiple output windows; attractive user interface with menus, buttons and text boxes; etc.

5 https://www.gnu.org/software/make/
6 https://maven.apache.org/
7 http://ant.apache.org/
8 https://gradle.org/

Most IDEs also support the integration of third-party software with quite important functionalities such as: version control libraries (e.g. GitHub[9], Apache Subversion[10]), plugins, full stack management, etc.

Typically the IDE is designed as a standalone tool that provides a graphical interface that allows the programmer to interact and automate the development in a very agile way and from a single Workspace. In spite of the different purpose, great diversity of parametrization and each IDE have a proper notion of environment, they are built so that the diverse tools work together to present a seamless development set for the developer.

Eclipse[11], Visual Studio[12], IntelliJ[13], Android Studio[14], XCode[15], among others, are some cases that have most of the presented features. In Subsection 5.5.1 are some examples of IDEs as well as some details about how they work.

5.1.2 *Advantages and Disadvantages*

The overall goal and main benefit of an IDE is to improve developer productivity. Its utilization by developers, rather than the existing alternatives, has many interesting advantages such as (Veracode, 2018):

FASTER AND BETTER PROJECT SETUP

Without integrated support, the whole process of initializing and configuring a project can be quite painful. With this method it is possible to get some basic configurations from scratch and manage some functionalities through a single application without having to walk between different tools and workspaces. When well chosen, it will require a minimal effort to the programmer in the setup phase and this becomes even more relevant when it is still at a beginner level because it takes away the burden of learning in detail a series of technologies simultaneously and allows it to focus only on the essentials.

FASTER AND BETTER DEVELOPMENT TASKS

Most of the IDEs features are presented so that the development process became more agile and uniform. Display of diagrams and other types of efficient resources management, real-time feedback of code errors through automatic parse and syntax checking as the code is being edited, automatic creation of pseudo-code (e.g. cycles structure) or code which may be inferred (e.g. gets and sets in Java), providing visual

9 https://github.com/
10 https://subversion.apache.org/
11 https://www.eclipse.org/
12 https://www.visualstudio.com/
13 https://www.jetbrains.com/idea/
14 https://developer.android.com/studio/index.html
15 https://developer.apple.com/xcode/

and keyboard shortcuts that reduce the required steps and clicks to obtain results, are some of the factors that make it possible to program more and with less effort using IDEs.

FASTER AND BETTER PROJECT MANAGEMENT AND COLLABORATION

Because of the IDEs inherent integration, programmers are forced to think of the project more globally in terms of the entire development life cycle rather than a series of discrete tasks. They also encourage and simplify the use of code comments for the programmer and provide some automation of documentation as well automatic generation of reports and other visual resources. It also benefits the joint work of programmers in a single work tool.

ENFORCE PROJECT OR COMPANY STANDARDS

The use of the same development environment interface by a group of developers standardizes the development process and allows to smooth and accelerate the entry of a new team member. Standards can be further enforced if the IDE allows the inclusion of custom templates and sharing them between members/teams working on the same project.

CONTINUAL LEARNING

The use of IDEs is also a great way for a developer to keep informed about the latest area practices, features and trends. Instead of being comfortably attached to them favorite tool chain and text editor for a long time, they are compelled to stay current and experiment new concepts and tools to take more advantage of them and increase their productivity.

Naturally, some drawbacks are also pointed out in the use of this developing model. There are IDEs for each degree of user knowledge and an inadequate choice will cause the programmer, for example, to get lost in the middle of excess information and features that he does not yet want or perceive. Also the need to learn the various aspects inherent to an IDE requires an initial investment of time and patience until the greater efficiency in the developing is achieved. It is also true that an IDE is a strong weapon but will not fix bad code, practices, design or performance and that in the hands of someone with less solid programming bases facilitates the creation of heinous code but still runs. So it's up to the programmer to have some knowledge of what he's doing and make right decisions to make good use of what's being provided to him.

Each IDE has its own niche strengths and weaknesses that differentiate them and make them more or less interesting according to some factors such as target public or project purpose. However, from the various aspects presented above and comparing with the existing alternatives, we conclude that with the adjusted choice of an IDE the disadvantages pointed out are easily overcome in the medium term and the gains are immense. Although

IDEs are not required to program, we believe that they have very useful tools to support any project and are very beneficial to increasing developer productivity.

5.1.3 *Differentiation Factors*

Although at first glance most of the IDEs seem quite generic and similar to each other because of the many common aspects they possess, in practice they all turn out to be different in some important criteria due to the options taken during their design. How to maximize user productivity, improve target audience's usage experience, make the product more financially profitable, the user's experience level expected, among others.

IDEs are available from Open Source communities, vendors and software companies; different pricing and licensing (e.g. totally free, free depending on the type usage, paid for a total license amount); different target audiences (e.g. beginners, advanced, students, professionals); supported operating systems (Windows, Linux, macOS, etc., or even several simultaneously); system model (e.g. standalone, part of a suite of compatible applications, plugin, Cloud Service); different target machines and purpose (mobile, embedded system, databases, web, etc.); supported programming languages (e.g. specific for just one language, one paradigm or even multiple-languages); features that provide (e.g. profiler, static code analysis, GUI builder); whether or not to include plugins and extensions (free or marketed); etc.

Although not all of them have the same relevance, they nevertheless end up being the main reason for adopting or rejecting the product by programmers and interfering with other relevant factors such as the type of user or company that will use them, the machine and target application, time and cost of use, among others.

5.1.4 *Summary*

Tracking the IDEs evolution, also the source code editors has evolved substantially making the line that separates them more and more tenuous. Sublime Text[16], Atom[17] and Visual Studio Code[18] are some examples of very sophisticated choices in terms of customization and integration of plugins and ultimately few are the aspects that differentiate them from true IDEs. They are therefore also great options for any developer who prefers a more personalized environment with the added cost of a greater effort in the setup of the whole system in order to enjoy more options than the just normal source code editing.

16 https://www.sublimetext.com/
17 https://atom.io/
18 https://code.visualstudio.com/

More and more models beyond the usual standalone application are gaining interest from developers, such as Software-as-a-Service (SaaS) and namely Cloud IDEs. This service allows to use the web browser as a client and to access a good range of cloud-based applications and services. Although there is still some distrust from developers in the adhesion to this model, in fact owns quite interesting advantages such as: requires virtually no download or installation, compatible with a greater number of devices, access to software development tools from anywhere in the world and easy collaboration between people in different locations. Once this model has the integration of more tools and taking into account the great potential that presents, as well as the advantages enumerated and the great adhesion to services in the Cloud by the companies, this may well be the model in vogue in the next decade for the software development industry (Veracode, 2018).

IDEs clearly make the software development process much more simplified by providing useful tools for all tastes and needs. With just one workspace it is possible to edit, compile and debug code in such a clear and natural way that sometimes developers do not even realize that exist different phases. Choosing an IDE that fits properly the programmer and project is very important because it increases the comfort and productivity even more significantly.

5.2 COMPILATION PROFILES

One of the most interesting aspects of IDEs is that they allow to create and select different environment profiles. This feature allows the user to automate and streamline part of the development process, providing a set of predefined configurations for different types of projects, builds, users, application to develop, among others. Profiles are so present in IDEs that sometimes users do not even realize that they are using them when they get an output from an application in a more detailed way or when they build the entire project by selecting only one shortcut. The type, quantity and level of sophistication of the provided options vary greatly from IDE to IDE. However, its benefits are quite considerable and indispensable for any user, regardless of their type of use or level of knowledge.

One of the compilation strategies practiced by IDEs is precisely a feature with these characteristics, namely compilation profiles. The IDE suggests to the user one or more ways to compile his programs without having to perform a great interaction (in most cases just one click) or need to define any compilation steps or parameters. The recommendations may vary in quantity and complexity, being generally presented in the form of optimized profiles for different development phases such as Debug or Release. It is rather curious to note that although several profiles with the same name and purpose are found for different IDEs,

the optimizations they recommend are mostly different, proving to be very interesting to analyze what each IDE intends with each optimization flag and its real energy impact in the executable generated.

Although all the studied IDEs share this compilation strategy, they vary in the way the profiles are presented as well as in their quantity and sophistication. Considering only the compilation profiles objectives (mainly the presence or absence of optimizations), it is possible to classify the analyzed IDEs in three groups:

NOT OPTIMIZED PROFILES

Set of IDEs that only provide the user with a default profile and without care regarding the optimization of the resulting code. The user can also compile the program comfortably but only in the simplest and inefficient way if they choose the provided default parameters. These IDEs are generally very simple and lightweight, usually used by those who are taking the first steps in the area and looking only for some work tool that automates the minimum necessary.

OWN PROFILES

More sophisticated IDEs that develop their own profiles with optimized compile parameters. They present two or more that they consider relevant for a given development phase or the purpose of the target application. Being Debug and Release the most common profiles, it is also possible to find others with more specific goals such as reducing the size of the resulting code or the level of debug information in the various stages of development. This group of IDEs also stand out because they offer easy exchange between profiles and also options with enough information for the user to create their own compilation profiles.

BUILD AUTOMATION TOOLS PROFILES

In convergence with the two previous groups, there are IDEs that although they choose not to design custom optimization profiles for their tools, yet provide similar functionality through the strong integration of BATs. These tools are quite powerful and allow through a scripting language to execute and validate quite a set of tasks, reducing the space to possible problems. The automation of compilation is one of the most solid strands they present, demonstrated through the extensive list of features they provide.

IDEs from this group choose to delegate to external tools part of the definition of the compilation process, namely the choice of the optimization parameters of the compilation profiles. This option allows them to inherit all the advantages that the initial tools offer as well as increase the portability of the projects. However, because the

use of these tools can be done externally or even integrated by other IDEs, this option reveals itself, although quite efficient, less personalized and more generic because the code generated will be precisely the same for all who use them.

Naturally, they all allow the user to manually change the build parameters. Nonetheless, it is very interesting to analyze the way of thinking of each IDE, what decisions they make, what standard options they provide and how optimized they are.

5.3 THE COMPUTER LANGUAGE BENCHMARKS GAME

Naturally, each programmer has his own level of knowledge and programming style allowing the existence of multiple software solutions for the same programming challenge. This diversity of solutions, which is already significant itself, easily expands according to the challenge complexity level or if different technologies are used (for instance different programming languages or paradigms). Consequently, the need arises to analyze the multiple solutions and compare them taking into account several factors relevant in this matter, such as execution speed, amount of resources used, level of abstraction, energy consumption, solution size or parallelization capability.

The Computer Language Benchmarks Game[19] (CLBG) is precisely a software project that compares implemented software solutions in most popular programming languages. The project contains a set of very simple but diversified algorithmic problems that work as challenges for their community. This follows as a game in which any user can consult or submit a solution in the technologies of his choice (e.g. select the language, operating system or even the compiler/interpreter) and see it evaluated and compared with the remaining ones. After a validation process, all submitted programs are measured on the same target machine and using the same measurement process. The CLBG also provides a framework that allows for the user to validate and test the submitted solutions according to some criteria, and also the possibility to measure and compare them in terms of CPU time, elapsed time, memory used and the gzip[20] size of the source code. They also provide a website with all the information previously mentioned as well as rankings, graphs and other comparative illustrations for all problems and languages involved along with other useful notes from the developer team (Gouy, 2018b).

The project content as well as its objectives have been transformed and developed over the more than 15 years of existence. Initially the main objective was only to compare all the major scripting languages, but progressively it was growing and today it already includes

19 https://benchmarksgame-team.pages.debian.net/benchmarksgame/
20 https://www.gnu.org/software/gzip/

languages of several paradigms (e.g. functional, imperative, object-oriented), types of execution (e.g. compiled, interpreted, virtual machine - and including several versions and manufacturers), multiple operating systems, etc. Nowadays the analysis performed focuses mainly on three aspects related to the solutions developed: which are the fastest, which are the most succinct and which are the most efficient. Each time a new version of a language or other technology is used, all programs affected are remeasured and those that do not pass the validation tests are removed. Currently, the project encompasses 10 active challenges (and more 5 that are obsolete/removed) and more than 1100 valid solutions covering more than 20 different operating systems. Each challenge has solutions in between 19 and 33 different programming languages in a total of over 40 (Thiel, 2018)(Gouy, 2012)(Gouy, 2018a)(Gouy, 2018b).

This project has gained some interest from the scientific community especially since 2009. Although the creators did not have that intention, since the project presents such simple and diverse problems, such heterogeneous software solutions and analysis of relevant factors, it has proven to be an excellent source for a considerable amount of scientific studies (Williams et al., 2010; Wrigstad et al., 2010; Shirako et al., 2009; Karmani et al., 2009; Brunthaler, 2010; Gerakios et al., 2010; Pestov et al., 2010; Homescu and Şuhan, 2012; St-Amour et al., 2012; Li et al., 2013; Sarimbekov et al., 2013).

From our point of view, the project proves to be a very challenging and didactic game for any programmer and at the same time an excellent source of benchmark software for other types of analysis. Despite the divergence of objectives between the CLBG and the study intended here, we consider however that its content may be useful for our investigation. As already mentioned, CLBG focuses on the comparison of runtime performance of software solutions implemented in multiple languages and operating systems. Although they do not take into account energy issues, we believe that our results may add further value and complement very well this limitation, an approach already adopted in other studies (Pereira et al., 2017; Lima et al., 2016; Couto et al., 2017; Oliveira et al., 2017). We also consider that although we are confined to only the C language, the GCC compiler and a Unix system, the solutions contained in the project remain quite relevant to the intended analysis. Opting to choose multiple IDEs and restricting some of the other variables, allows us to obtain more secure data about them and to keep the focus on their analysis and their compilation profiles.

5.4 EXPERIMENTAL SETUP

5.4.1 *Testing Platform*

The study was accomplished on a laptop Asus N56JN-DM127H, running under Linux. The hardware/software resources most relevant characteristics for the required analysis are: Ubuntu 16.04.4 LTS 64-bit (Linux Kernel 4.13.0-36); Intel® Core i7-4710HQ up to 3.5 GHz, Haswell Family; 8 GB DDR3L 1600MHz; and NVIDIA® GeForce® GT 840M, 2GB DDR3 VRAM.

5.4.2 *Measurement Software*

In order to obtain the desired criteria for the experimental analysis, the framework developed for the study presented in Chapter 4 was used again. Briefly, this is a C program that uses the RAPL interface to read information about energy consumption of components from machine registers. The tool also allows a greater set of functionalities that aid the mentioned process and potentiate other types of approaches according to the needs of the user.

In the case of this particular study, the tool allows to measure by software with a high degree of certainty the energy consumption of CPU and memory during the execution of a given program. Obtaining this type of information and its processing is undoubtedly fundamental to complement our analysis in order to have concrete experimental data that we can address. The versatility of the tool is also an important factor as it allows exploring different methodologies to be applied in this study. In fact, the wide range of options reduces possible restrictions in the methodology design and allows the tool to adapt to it and not the other way around.

5.4.3 *Measured Software*

After investigation of several possibilities of target programs for this study, we decided to use the CLBG project to obtain benchmarks for the accomplishment of the desired measurements and analysis.

The programs contained in that project prove to be an excellent set of options that perfectly fit the needs and objectives of this study. Firstly, because they respect essential requirements for our project such as the software be open-source, have no graphical interface, runtimes of less than 1 minute and work properly on Linux and with GCC. Moreover, they have other very interesting and challenging properties such as being from different au-

thors, background areas and application fields making them quite diverse from each other; they are highly optimized solutions making them even more challenging both at the level of resources used and in terms of their compilation/execution process; have few external dependencies and do not need access to any type of network, allowing to reduce the external noise and the complexity inherent to the installation/measurement process (as well as making it easier to focus on them); they have runtimes that allow us to perform a greater number of consecutive measurements and avoid possible overflows during the measurements; among other factors.

12 programs contained in the CLBG were selected. Below, a brief description, according to the project authors, will be presented as well as the version used in this study and which inputs were considered.

FASTA

Generate and write random DNA sequences by copying from a given sequence and by weighted random selection from 2 alphabets.
Version selected: 2; Input used: 25000000.

N-BODY

Double-precision N-body simulation. Model the orbits of Jovian planets, using the same simple symplectic-integrator.
Version selected: 4; Input used: 50000000.

FANNKUCH-REDUX

Indexed-access to tiny integer-sequence, as defined in "Performing Lisp Analysis of the FANNKUCH Benchmark"[21]. For a given n that tends to infinity, it is conjectured that the count approaches at most n*log(n).
Version selected: 5; Input used: 12.

SPECTRAL-NORM

Eigenvalue using the power method. Based on the statement in point 3 of the set of challenges published in 2002 by SIAM News called "Hundred-Dollar, Hundred-Digit Challenge Problems"[22][23].
Version selected: 5; Input used: 5500

MANDELBROT

Generate Mandelbrot set portable bitmap file. Based on the mathematical problem called Mandelbrot Set[24], which in general terms is a particular set of complex numbers

21 http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.5124
22 http://mathworld.wolfram.com/Hundred-DollarHundred-DigitChallengeProblems.html
23 http://www.siam.org/siamnews/01-02/challenge.pdf
24 http://mathworld.wolfram.com/MandelbrotSet.html

which has a highly convoluted fractal boundary when plotted.
Version selected: 6; Input used: 16000.

REGEX-REDUX

Match DNA 8-mers and substitute magic patterns. In this particular case, through the use of the same simple regex patterns and actions to manipulate FASTA format data.
Version selected: 4; Input used: fasta file input25000000.txt.

K-NUCLEOTIDE

Hashtable update and k-nucleotide strings. Requires mapping of DNA letters and the use of hash functions (built-in or library) that concatenates those codes is an acceptable optimization.
Version selected: 1; Input used: fasta file input25000000.txt.

REVERSE-COMPLEMENT

Read DNA sequences and write their reverse-complement from the sequence of bases of each strand.
Version selected: 6; Input used: input5000000.txt.

BINARY-TREES

Allocate and deallocate a large amount of perfect binary trees, using a simplified adaptation of the method of Hans Boehm's GCBench[25].
Version selected: 3; Input used: 21.

CHAMENEOS-REDUX

A peer-to-peer cooperation paradigm adapted from what is stated in "Chameneos, a Concurrency Game for Java, Ada and Others"[26].
Version selected: 5; Input used: 60000000.

METEOR

Algorithmic search of solutions for the Meteor Puzzle (10x5).
Version selected: 1; Input used: 2098.

THREAD-RING

Simplistic adaptation of the process described in "Performance Measurements of Threads in Java and Processes in Erlang"[27] and "A Benchmark Test for BCPL Style Coroutines"[28], where messages are passed between N threads/processes that are spawned connected as a ring structure.
Version selected: 3; Input used: 5000000.

25 http://hboehm.info/gc/gc_bench/
26 http://cedric.cnam.fr/PUBLIS/RC474.pdf
27 http://archive.is/1droG#selection-129.1-129.204
28 http://www.cl.cam.ac.uk/ mr10/Cobench.html

The CLBG project provides in its online platform much more information about the programs presented as for example some more background on what inspired their challenges, more detailed explanation about the behavior of the intended algorithm (including recommendations and restrictions of implementation), practical examples of some supported approaches, suggestion of inputs and respective outputs generated (namely which will be considered valid for participation in the project game) and previous results of measurements submitted for different platforms and languages.

With the exception of *pidigits*, all the programs that are considered active for the CLBG are used. This exception is due to technical difficulties because unfortunately only one C version of the program is available and it was not possible for us to get it working correctly. The three programs considered obsolete by the CLBG (*chameneos-redux*, *meteor*, *thread-ring*) are also included in our study, as they are still perfectly valid and have value for our analysis.

The vast majority of the challenges, besides covering a wide range of programming languages, also have several solutions implemented for each specific language. Since C is not an exception, there are several versions that can be considered for our study. We chose to use the version better positioned in the performance ranking provided by the CLBG for each problem because we consider that the higher sophistication they present could bring more relevance to the results obtained.

Regarding the inputs selected, preference was given to the values recommended by the CLBG itself to test the programs performance. However, the order of magnitude was changed for some cases in order to reduce the size of the output generated or to increase the execution time to values greater than only a few seconds. With these improvements on the one hand it has become feasible to perform the large number of measurements we wanted without generating terabytes of disk information and on the other hand make the execution time of the programs more homogeneous and to make more salient the possible effects caused by the parameters considered. For all analyzed cases the input is fixed for all executions of a given program and only has one of two formats: a natural number or a .txt file generated from the *fasta* program.

It was not necessary for us to make substantial changes in the source code of the programs analyzed.

All the programs were prepared to receive as an argument a variable element that defines, among other aspects, the dimension of execution. Taking into account the option of using a fixed input for each program, a hard coded variable was introduced in all programs with

their respective input value. This decision also makes it easier to automate the measurement process.

For the *mandelbrot* and *spectral-norm* programs it was still necessary to apply an extra change. It was found that the source code collected did not compile with the -Oo suite due to conflicts with the inline declaration of some functions. This limitation was overcome with the inclusion of the static keyword in the respective functions, being a procedure that is already established in the *binary-trees* program (which also served as inspiration). This approach does not present any limitation to the functioning of the program itself since the mentioned functions are at no time invoked externally (GCC team, 2018a).

Neither of the changes made alters the result of the program or any other aspect of the original operation of its algorithms. Therefore, in its essential, all the analyzes and measurements made can be seen as if the programs had entirely in their original version for a certain fixed input.

Analyzing the CLBG ranking that includes all versions submitted for the most diverse languages and programming platforms, the versions considered in this study are quite optimized being highlighted in the respective tops of each challenge. They are mainly in the first two positions in terms of runtime performance and are equally well positioned in the other factors considered by that project. This fact further emphasizes the value of the solutions used as well as making the choice of the CLBG even more relevant as the source of benchmarks for our study.

5.5 METHODOLOGY

5.5.1 *Analyzed Tools*

Integrated Development Environments

Taking into account the previously established study parameters (in particular the machine used and the intended analysis) as well as other external factors such as the great diversity of IDEs in the current market, some selection criteria were defined for the candidate study tools in order to obtain more relevant results. A given IDE is considered a candidate to be analyzed by our study if it complies fully with the following requirements:

- Running under Linux environment;
- Support the C language and GCC compiler;
- Have a stable and release version;

- Capable of accomplishing the tasks under study without resorting to plugins installation;
- No cost of usage or at least have a trial version;
- Specification of the parameters used during the compilation process.

For the study, 41 IDEs supporting C/C++ language were preliminarily collected as possible candidates. This collection was formed based on several research, including several online listings and blog posts regarding interesting criteria such as good quality and great user acceptance. Due to the fact that they did not meet at least one of the criteria required for the study, 26 options were discarded:

- Tools that only have alpha version in Linux or because they are discontinued: Bloodshed Dev-C++[29], Philasmicos Entwickler Studio[30] and Squad: Collaborative IDE[31];
- Dedicated to C++ or not support C/C++ without plugins: Ultimate++[32], IBM Rational Software Architect[33], MonoDevelop[34], OrionHub[35];
- Only compatible with Windows, MAC, iOS or Android: C-Free[36], SkyIDE[37], Open Watcom[38], Microsoft Visual Studio[39], Bricx Command Center[40], MinGW[41], LccWin32[42], Pelles C[43], Xcode[44], AppCode[45], C++Builder[46];
- Does not provide free access: Digital Mars IDDE[47], LabWindows/CVI[48], SlickEdit[49], Understand[50];

29 http://www.bloodshed.net/devcpp.html
30 https://www.philasmicos.com/index.php
31 https://squadedit.com/tour
32 https://www.ultimatepp.org/
33 https://www.ibm.com/developerworks/downloads/r/architect/index.html
34 https://www.monodevelop.com/
35 https://orionhub.org/
36 http://www.programarts.com/cfree_en/
37 http://www.skyide.com/
38 http://www.openwatcom.org/
39 https://www.visualstudio.com/
40 http://bricxcc.sourceforge.net/
41 http://www.mingw.org/
42 https://lcc-win32.en.uptodown.com/windows
43 http://www.smorgasbordet.com/pellesc/
44 https://developer.apple.com/xcode/
45 https://www.jetbrains.com/objc/
46 https://www.embarcadero.com/products/cbuilder
47 https://digitalmars.com/features.html
48 http://ni.com/cvi
49 https://www.slickedit.com/
50 https://scitools.com/

- Does not mention or disclose how the compilation process is performed: Repl.it[51], CodePad[52], JDoodle[53], CodeChef[54].

Some of the mentioned tools fail even several of the required criteria, but for simplification purposes they are only associated with one of the requirements listed above. It was observed some balance in the amount of candidates removed per requirement, with the exception of support in Linux that contributes to the discard of about half of the candidates.

A common aspect in the candidates who were dispensed due to the last criterion presented is the fact that they are all Cloud IDEs. It has been found that this kind of tool usually even indicate to the user some informative elements such as which version of GCC is made available or data related to the submitted program (such as runtime and memory consumed). However, they omit other relevant information such as what compilation parameters are used to generate the requested executable.

There was an effort on our part to get more information from the tools in question, but it was not possible to get any clarification on the subject matter (in some cases not even an answer). We were only informed by some that in the future they will provide several compilation profiles, but we were not told which parameters they use at the moment (which is one of the most relevant information for our study). In general, the majority of Cloud IDEs focus mainly on Web programming, providing more information and directing the more advanced features for this type of technologies. At the moment there is still some lack of consideration in aspects that we consider relevant in the scope of our study.

Table 8 presents the 15 IDEs that were found in our research and considered valid for analysis in this study: CLion[55], NetBeans[56], Code::Blocks[57], CodeLite[58], Eclipse CDT[59], KDevelop[60], Geany[61], Anjuta DevStudio[62], Qt Creator[63], DialogBlocks[64], ZinjaI[65], GPS[66],

51 https://repl.it/
52 http://codepad.org/
53 https://www.jdoodle.com/
54 https://www.codechef.com/
55 https://www.jetbrains.com/clion/specials/clion/clion.html
56 https://netbeans.org/
57 http://www.codeblocks.org/
58 https://codelite.org/
59 https://www.eclipse.org/cdt/
60 https://www.kdevelop.org/
61 https://www.geany.org/
62 http://anjuta.org/
63 https://www.qt.io/
64 http://www.anthemion.co.uk/dialogblocks/
65 http://zinjai.sourceforge.net/
66 https://www.adacore.com/gnatpro/toolsuite/gps

IDE Name	Studied Version	Usage Model	License Type	Target Audience
Code::Blocks	17.12-1	Standalone	Free License	Beginner
Geany	1.33	Standalone	Free License	Beginner
DialogBlocks	5.15.3 (Unicode) Built Dec 13 2017	Standalone	Free for unregistered and registered account	Beginner
ZinjaI	20180221	Standalone	Free License	Beginner
Anjuta DevStudio	3.18.2	Standalone	Free License	Intermediate
GPS	20170515-63	Standalone	Free License	Intermediate
CLion	2018.1 Build #CL-181.4203.54	Standalone	Free Trial	Advanced
NetBeans IDE	8.2 Build 201609270201	Standalone	Free License	Advanced
CodeLite	12.0.0	Standalone	Free License	Advanced
Eclipse CDT	9.4.3.201802261533	Standalone	Free License	Advanced
KDevelop	5.2.1	Standalone	Free License	Advanced
Qt Creator	4.6.0 Based on Qt 5.10.1	Standalone	Free open source version and trial commercial version	Advanced
Oracle Developer Studio	12.6	Standalone	Free after account registration	Advanced
Sphere Engine	20180319.r445	Cloud	Free Trial	Intermediate
AWS Cloud9	Not specified	Cloud	Free with limited resources	Advanced

Table 8.: Analyzed IDEs.

Oracle Developer Studio[67], Sphere Engine[68] and AWS Cloud9[69]. They were all installed in the machine previously described and were tested the most recent versions available until the 29th of March. We consider that it is a very interesting set for the intended analysis, highlighting its considerable scope and diversification in aspects such as its sophistication, usage type, business model, target audience and number of languages natively supported, among others. Interestingly, all tools are cross-platform.

Another differentiating factor observed among the selected elements is the diversity and quantity of languages and technologies that they support. On the one hand there are IDEs that stand out because they are quite generic and comprehensive in relation to that aspect (e.g. NetBeans, KDevelop) and on the other there are more restricted tools being even

67 https://www.oracle.com/tools/developerstudio/index.html
68 https://sphere-engine.com/
69 https://aws.amazon.com/pt/cloud9/

sometimes quite specific versions for a particular programming family or language (e.g. CLion or Eclipse CDT for C/C++).

We believe that this fact may have some relevance in the user's interaction with the tool and in the amount and quality of features it provides. Naturally, it is expected that the more specific the tool is, the more suitable and optimized it will be to work with a particular language. However, we consider that this differentiating factor will not have an impact within the objectives intended for our study because they are independent aspects. This observation can be analyzed in more detail in the final part of this study after concrete results were obtained.

In order to obtain the desired data from IDEs, the license type turn out to be indifferent for our analysis. None of them restricts the access of that kind of data, whether the license type is totally free, trial with limited time/resources, free for academic staff or open source version. Nonetheless, it is an aspect that has some inherent limitations to the tools (which differ greatly depending on the case and version itself) and which may be relevant to other areas of analysis, which has led us to choose to disclose the information regarding each one.

AWS Cloud9 and Sphere Engine differentiate themselves from the rest by having the Cloud IDE model as their utilization format, unlike the remaining 13 that are Standalone applications. In Figure 11 it is possible to observe an example of the common interface of a Cloud IDE (with the compilation options displayed). The Sphere Engine is also used or integrated in other similar online platforms (e.g. Ideone[70], Amplify[71], RecruitCoders[72]), allowing the results and analyzes obtained to this case may be extended to the other related tools.

70 https://ideone.com/
71 https://www.amplify.com/
72 https://recruitcoders.com/

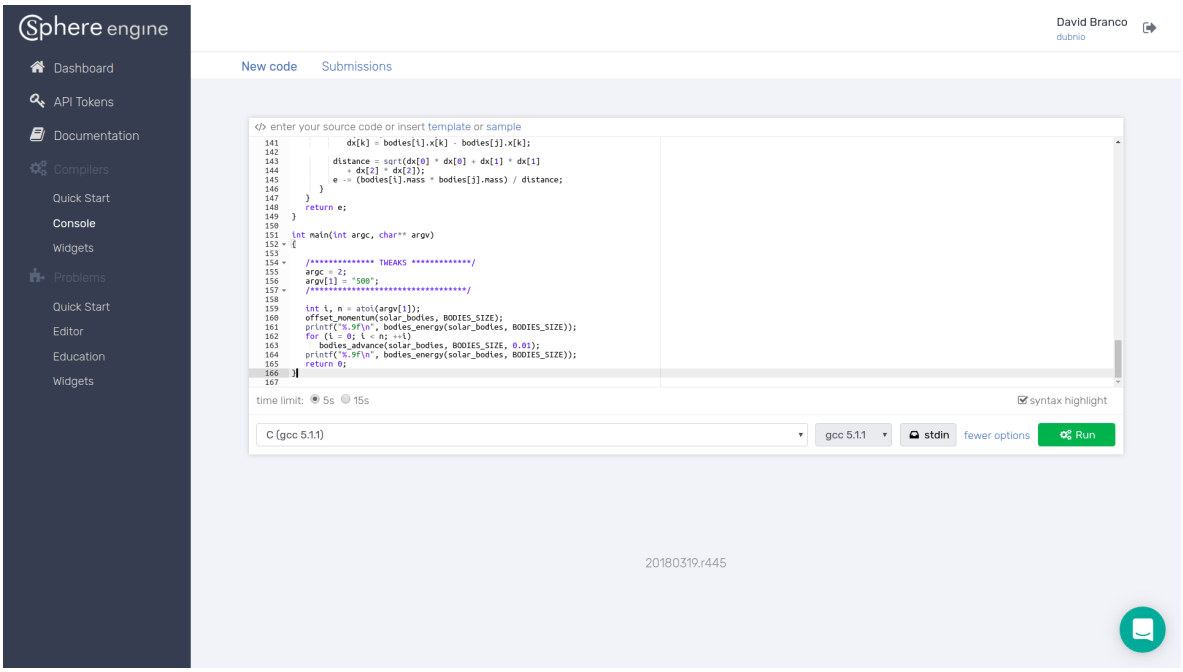


Figure 11.: Sphere Engine interface.

CLion, NetBeans, Eclipse CDT and Qt Creator are four of the most popular tools in the developers community for the languages and compiler concerned. They stand out by providing very important automation mechanisms for more demanding applications, ease of integration of plugins/build automation tools and for being a great help for large projects with multiple dependencies.

Although NetBeans is a very generic tool, it has a very interesting feature that is not considered by many tools. This IDE allows to add or remove specific packs for certain languages allowing the user to have a tailor-made IDE for their needs at the moment. Thus, the user do not need to work with the full version of the IDE if he just needs a particular paradigm, nor install a new IDE for every technology that they are interested in. This feature was even considered for this study where it was only installed the specific version for C/C++.

Geany, ZinjaI, DialogBlocks and Code::Blocks differ from the other Standalone applications for having as target audience mainly beginning users, in particular programming students. They provide a more simplified interface, with basic but quick and efficient mechanisms and without the need for large computational resources or external dependencies to operate (see Figure 12). The absence of advanced shortcuts and sophisticated features such as version control libraries or full stack management allow the user to focus more easily on

their purpose for the moment and to interact more lightly with the incidences of the code. Code::Blocks also allows in a very consistent way to integrate and expand the tool for other type of users and projects.

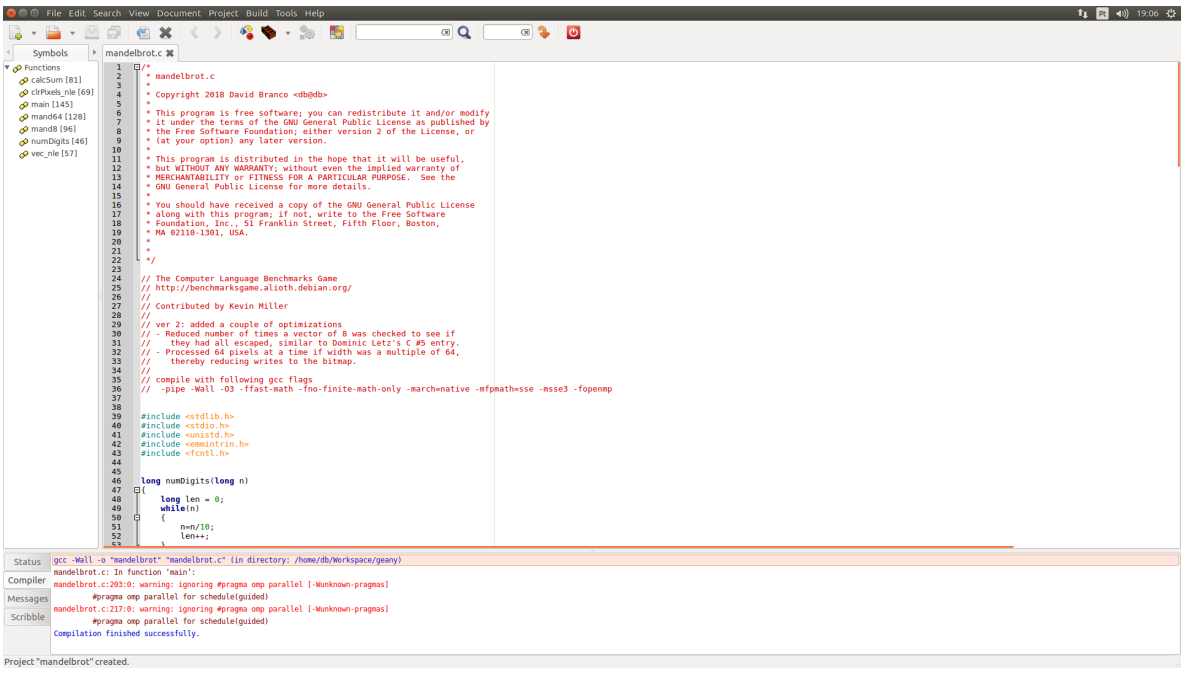


Figure 12.: Geany interface.

While we recognize the high sophistication degree that various source code editors currently have, we have decided not to include any for our analyze. It would be necessary to integrate plugins or libraries to perform the tasks required for this study and, as with IDEs, it is our intention to only analyze the standard versions provided to users without any kind of customization that may vary according to their preference. However, the study developed here as well as the results obtained can be easily related to those coming from more sophisticated source code editors, especially those that have the ability to integrate plugins with the same standard features.

Build Automation Tools

Applying the same criteria previously defined for the IDEs selections, were also analyzed 8 tools that are or have features of BATs. In particular, which have the ability to provide automatic generation of the necessary steps for the compilation of C/C++ projects. Despite there is a very popular set of tools with these characteristics, in fact it turns out that they are also conceptually quite generic since they are intended by nature to cover a wide range of

target applications. Although they support the languages and compilers that are intended for our study, most do not provide the user with predefined compilation parameters, requiring to fill in a skeleton provided with a representation of the desired values or to create construction environments from scratch.

For this reason, 5 of the candidates initially considered were discarded: Apache Maven[73], Gradle[74], Apache Ant[75], SCons[76] and Premake[77]. Naturally there are plugins and libraries that fill the gap presented (e.g. NuGet[78]), but as noted earlier the integration of these variants falls outside the context of our study.

Despite the limiting factors mentioned, it was possible to find in our research 3 tools that meet the required requirements and fit the intended objectives:

CMAKE

Powerful tool designed to build, test and package software. It provides the user with a wide range of predefined configurations that deeply embrace the many existing features as well as ease in customizing them. The user assembles in a single context-sensitive file the desired settings (CMakeLists.txt) and then generates all necessary steps for the project management such as importing components or generating output suitable for different operating systems, compilers or IDEs. Compared to the other tools presented below, this option stands out as being more generic, having a less user-friendly syntax and also for being more appropriate for projects with greater complexity and external dependencies. Eclipse, CLion, NetBeans, Sublime Text and TextMate are just a few examples of the many IDEs and Source Code Editors that integrate at least part of CMake's functionality into their products (CMake Reference Documentatiom).

QMAKE

Tool that allows with just a few lines of information to manage all the software build process in a very simplified way. It was designed and proves to be more appropriate for project management in Qt because it can automatically include build rules for MOC (the meta-object compiler) which provides signals and slots, or UIC (the UI compiler) which creates header files from .ui designer files and RCC (the resource compiler) which compiles resources. However, it has capabilities to intervene in projects external to the parent tool due to its great flexibility and high compatibility with other build systems such as Microsoft Visual Studio and Xcode (qmake Manual).

73 https://maven.apache.org
74 https://gradle.org/
75 http://ant.apache.org/
76 http://scons.org/
77 https://premake.github.io/
78 https://www.nuget.org/

QBS

> Qt Build Suite was originally designed from a list of suggested improvements presented in a blog post to be a Qmake replacement. It naturally has many similarities with the tool that inspired it (in particular the propensity for projects in Qt) but is currently already in a phase with some maturity and ideological independence. Optimized declarative syntax (QML subset), increased extensibility, fast incremental builds thanks to the project view as a whole and even greater ease in integrating external tools and transforming file types are some of the advantages that make Qbs, in some aspects, an optimized version of qmake (Qbs Manual).

For our analysis we used the versions 3.11.0 of CMake[79], 3.0 of qmake[80] and 1.4.5 of Qbs[81]. These tools have the ability to generate makefiles in a very simplified and configurable way that will later allow to build the entire project according to the user purpose. This feature is not present in all 8 analyzed tools and proves to be very pertinent since in addition to the obvious advantages to the user in terms of portability and dependency management, it allows to verify which parameters and standard instructions are passed to the compiler in order to create the desired executable.

Another interesting feature that they have in common is the great versatility of use allowing to work with these tools in multiple ways, with special emphasis through command line or integration into IDEs.

Through command line it is possible to check in detail the depth of options that these tools provide. However, only a small part of this set will be considered in this study, namely the support of compilation profiles that allows the user to automatically obtain compilation parameters, simply indicating through a textual (e.g. Debug, Release) or numeric (e.g. compiler version) representation the desired settings.

The integration of some functionalities of these tools into IDEs is another very relevant aspect because it allows to obtain the best of both worlds: obtain a familiar visual interface to the user and, at the same time, quite sophisticated features and independent of the environment used. The user can thus choose to import a file with the desired settings (created for example in another IDE or through command line) or yet to manage them through the menus and windows in the IDE itself.

Although the IDEs usually also have their own system for managing these aspects, it is however verified that some cases opt for a strong integration of very mature BATs and only create a graphical interface compatible, fully delegating to them the implementation of these functionalities. In some cases they even choose to integrate a vast set of tools, such

79 https://cmake.org/
80 http://doc.qt.io/qt-5/qmake-manual.html
81 https://wiki.qt.io/Qbs

as Qt Creator that integrates the 3 tools mentioned.

In total, among the 49 tools found 18 were considered valid according to the requirements and objectives of our study. In the following sections will be analyzed how the process of compiling a C program is carried out and what options exist in order to configure it, which compilation profiles each tool provides to the user, which represents in particular each of the parameters that compose them and what conclusions can be drawn from the collected information.

5.5.2 *Compilation Options*

GCC Overview

From an overall perspective, the compilation of a C program is a process consisting of four distinct phases: preprocessing, compilation, assembly, and linking (always in that order).

Briefly, initially is performed the preprocessing of the source code received as parameter. Then, is executed the actual compilation phase of one or multiple files in respectively one or more assembler input files (which are later transformed into object files). After the linker combines all the object files then the desired executable file is finally produced (Von Hagen, 2011) (Erlandsson, 2018). The whole process is also concisely illustrated in diagram 13.

Traditionally, C compilers orchestrate all this process by invoking other programs that handle each phase separately.

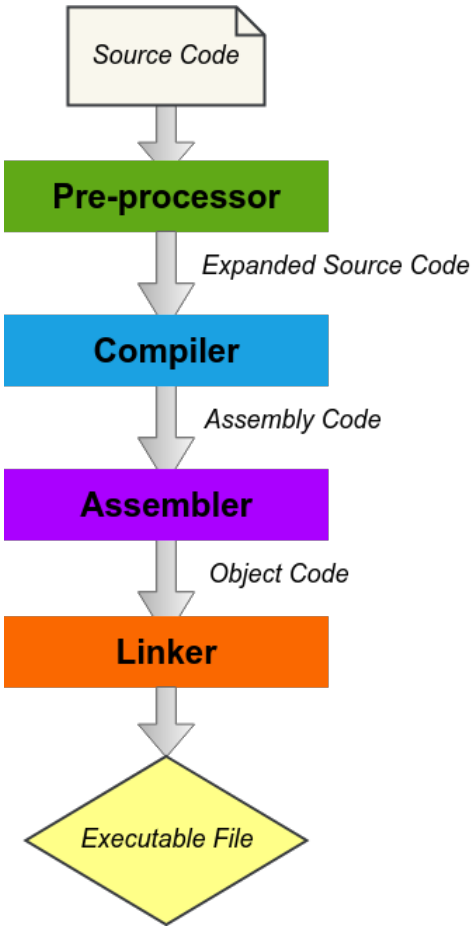


Figure 13.: Traditional C compilation stages.

GCC is a driver program that works precisely as presented. This tool in particular also allows some flexibility in relation to the compilation phases, allowing to control all or some part of the process by passing options as arguments when invoking the program. Some examples of this fact are the possibility to change the usual process flow, to execute or not a certain phase (e.g. the option -c indicates that the linker should not be executed), define the type of output generated, tweak for a different architecture and to optimize the code quality.

The configuration parameters passed to the compiler as an argument can have different formats, such as: single-letter (e.g. -L, -f), multi-letters (e.g. -dv, -fmove-loop-invariants), letters and numbers (e.g. -g3), deriving from the same suffix indicating the class of the same (e.g. -Wextra, -O2), have positive and negative version (for a -ffoo flag if it exists its negation will be in the form -fno-foo) or even have a restricted set of possible defined values (e.g. Wsuggest-attribute=[pure|const|noreturn|format|malloc], -fmax-errors=n). In addition to the predefined options, the program also accepts as argument other parameters

such as file names and directories. With a few exceptions and since the user pick options of different types, the order in which they are specified is not important for the final result (GCC team, 2018b).

It is also important to note that despite the great versatility of programming languages covered by GCC, in practice it is found that the vast majority of the options that are provided can be applied to the C language. The effective applicability of the flags in question already depends on several aspects such as target machine, operating system, source code, etc.

Briefly and according to the tool manual, the GCC parameters can be categorized as follows (GCC team, 2018b):

- **Overall Options**
  Controlling the kind of output: assembler files, preprocessed source, object files or executable.
  Examples: -c, -S, -E, -o.

- **C Language Options**
  Controlling the variant of C language compiled.
  Examples: -ansi, -std=standard, -fgnu89-inline.

- **C++ Language Options**
  Options controlling C++ Dialect.
  Examples: -fabi-version=n, -fno-access-control, -fcheck-new.

- **Objective-C and Objective-C++ Language Options**
  Variations on Objective-C and Objective-C++.
  Examples: -fconstant-string-class=class-name, -fgnu-runtime.

- **Diagnostic Message Formatting Options**
  Options to control diagnostic messages formatting.
  Examples: -fmessage-length=n. -fno-show-column.

- **Warning Options**
  Options to request or suppress warnings.
  Examples: -pedantic-errors, -Wall, -fsyntax-only.

- **Debugging Options**
  Options to produce debuggable code.
  Examples: -g, -fno-merge-debug-strings, -ggdb.

- **Optimize Options**
  Options which control code optimization.
  Examples: -falign-functions[=n], -fmerge-constants, -O3, -Ofast.

- **Instrumentation Options**
  Enabling profiling and extra runtime error checking.
  Examples: -p, -fbounds-check, -finstrument-functions.

- **Preprocessor Options**
  Controlling header files and macro definitions and also getting dependency information for Make.
  Examples: -traditional, -dD, -pthread, -MMD.

- **Assembler Options**
  Passing options to the assembler.
  Examples: -Wa,option, -Xassembler option.

- **Linker Options**
  Specifying libraries for linking.
  Examples: -s, -static, -nodefaultlibs.

- **Directory Options**
  Options for directory search (e.g. header files, libraries and executable files).
  Examples: -Idir, -no-canonical-prefixes, –sysroot=dir.

- **Code Generation Options**
  Specifying conventions for function calls, data layout and register usage.
  Examples: -fcall-saved-reg, -fno-jump-tables, -fverbose-asm.

- **Developer Options**
  Printing GCC configuration info, statistics, and debugging dumps.
  Examples: -fdump-tree-all, dletters, -fprofile-report.

- **Submodel Options**
  Target-specific options (such as processor variant).
  Examples: -march=name, -mapcs-frame, -mmcu=mcu.

Compilation Profiles

All the 18 examined tools (15 IDEs and 3 BATs) grant the user the possibility of automating the compiling process from their C projects using GCC through predefined compilation profiles.

The concrete way in which the profile values are described depends greatly on the tool in particular, being the most common through a log message in a program window (e.g. Figure 14 and 15), an interactive menu in the Build Section of the project's own settings (e.g. Figure 16 and 17) or by consulting the Makefile molded by the tool for the concerned project. In the particular case of the Sphere Engine that information is not provided together with the tool, but we were able to obtain it by email after a request. The change between profiles is usually performed in one of two ways: either by implied modification of a value in a configuration file (e.g. Figure 18 and 19) or through a visual shortcut in the tool main window (e.g. Figure 16 and 17).

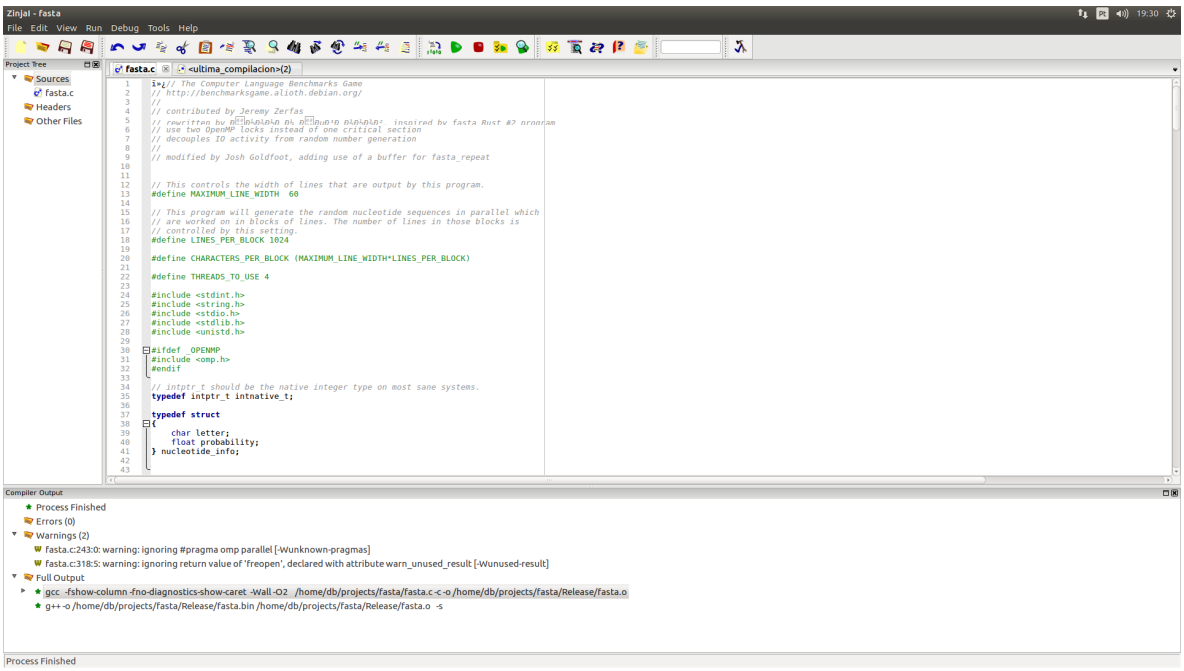


Figure 14.: Log message with the compilation parameters from ZinjaI.

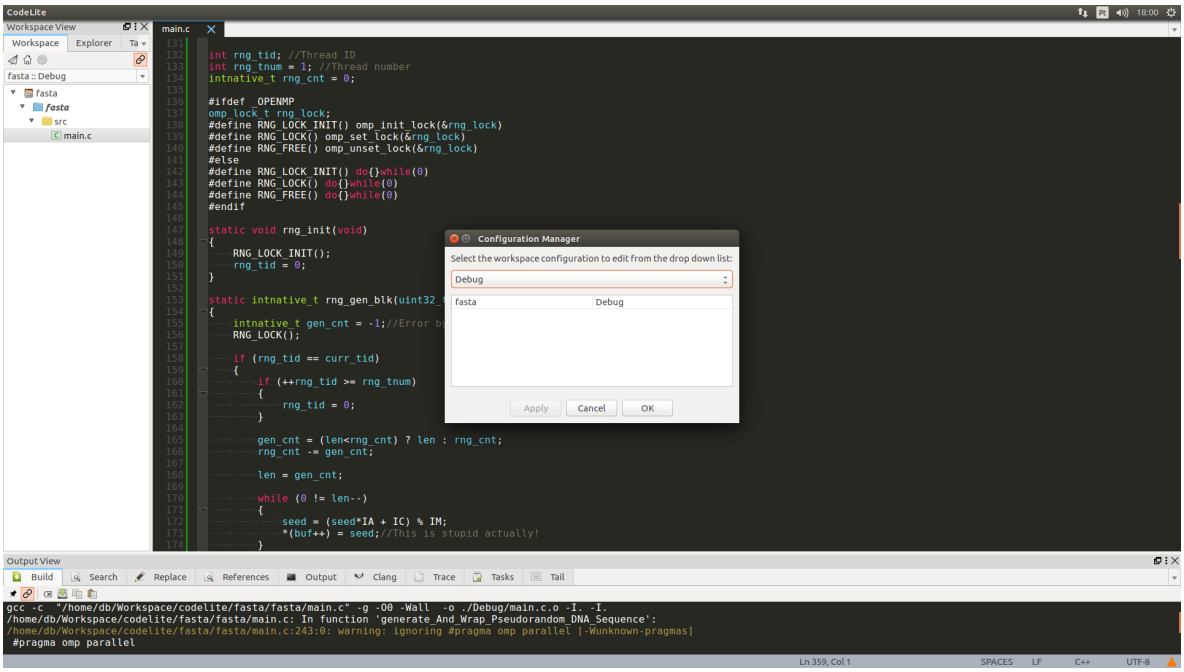


Figure 15.: Log message with the compilation parameters from CodeLite.

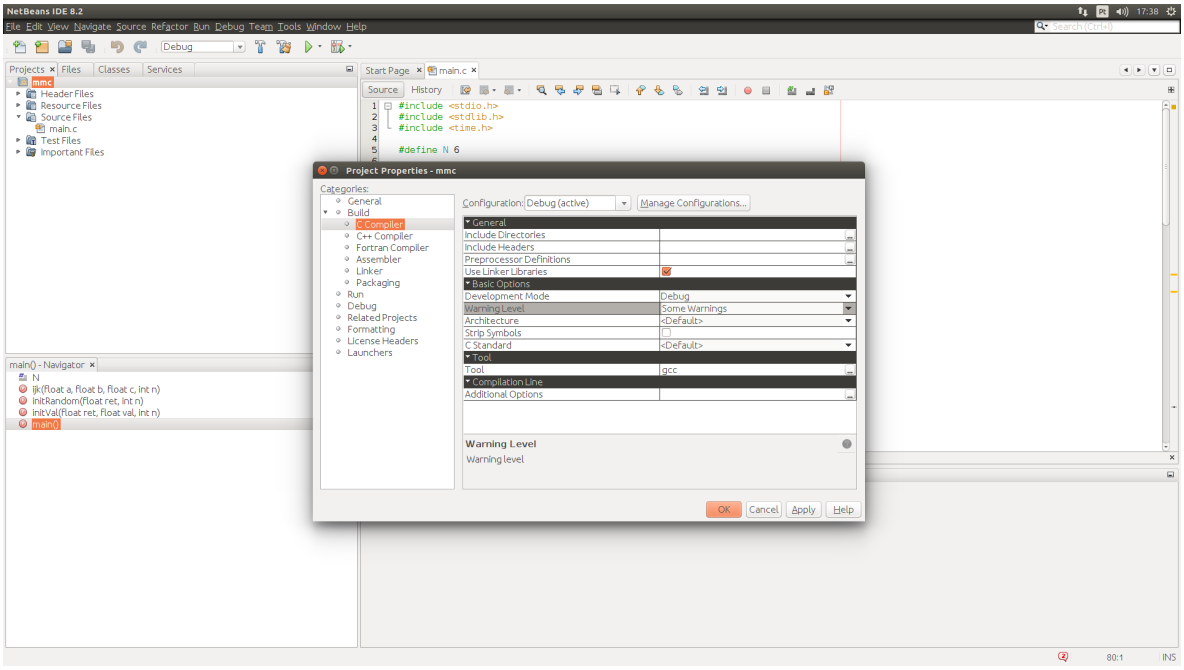


Figure 16.: Project properties manager from NetBeans IDE.

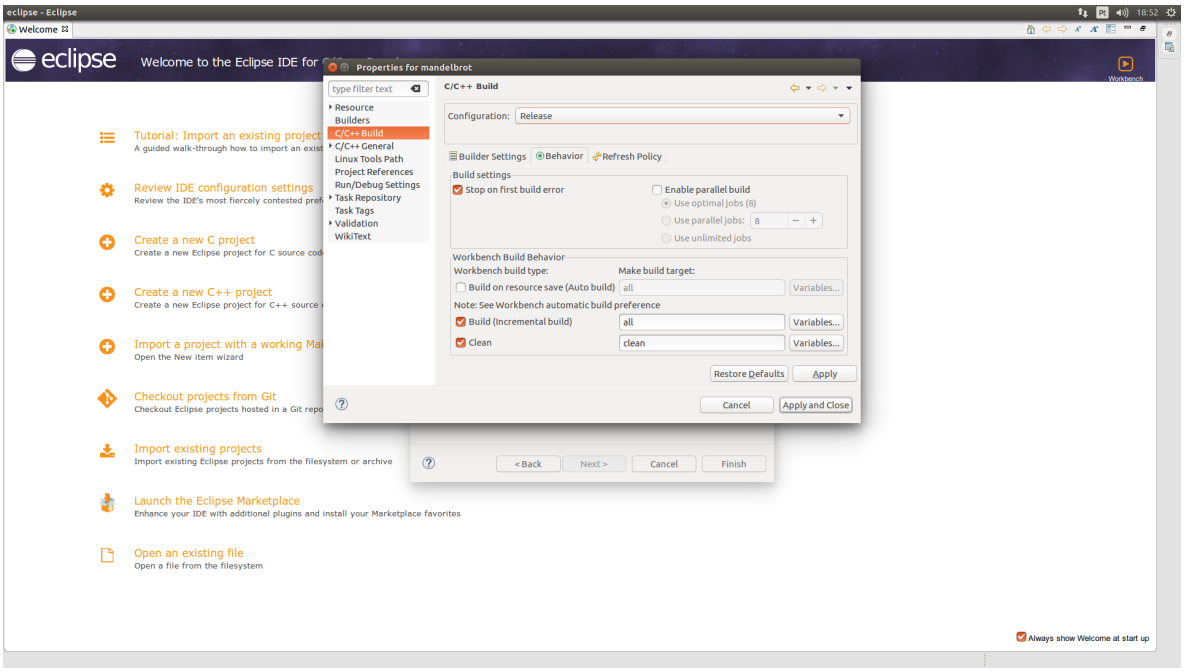


Figure 17.: Project properties manager from Eclipse CDT.

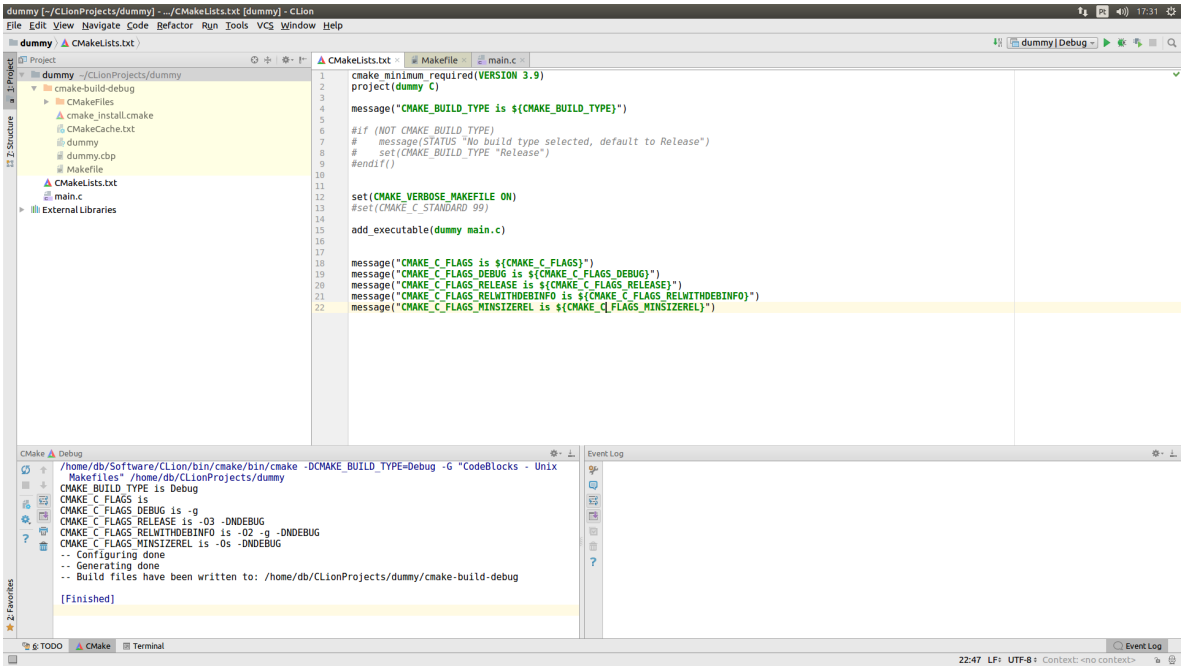


Figure 18.: Example of CMakeLists.txt from CLion.

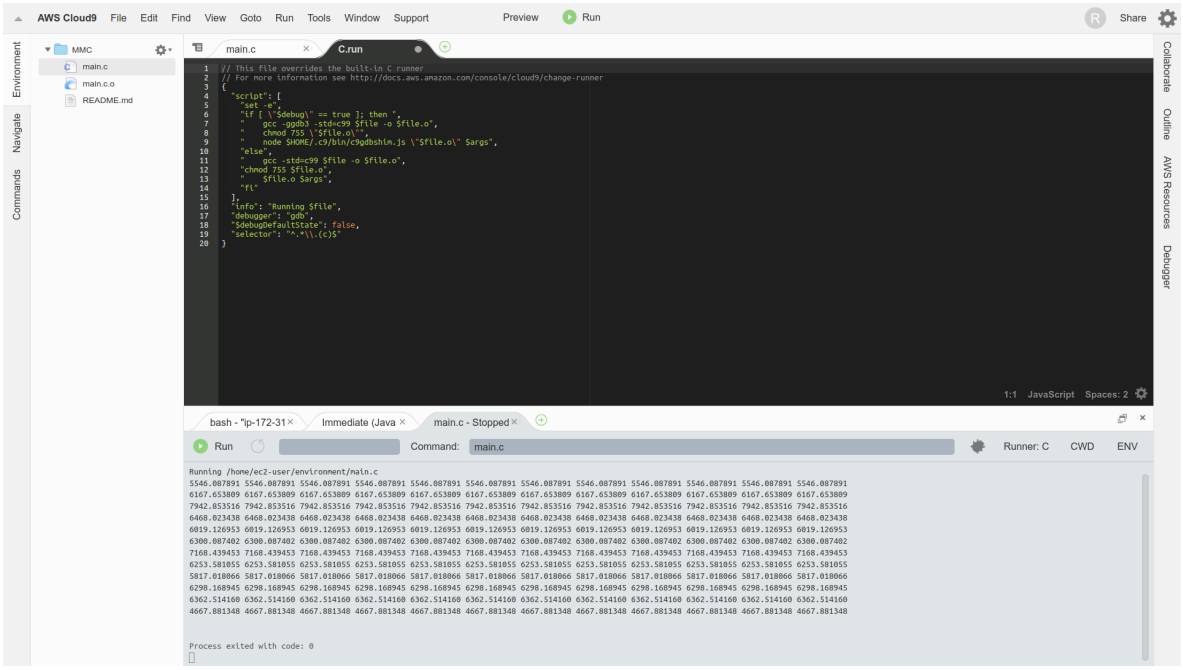


Figure 19.: Default configuration file from AWS Cloud9.

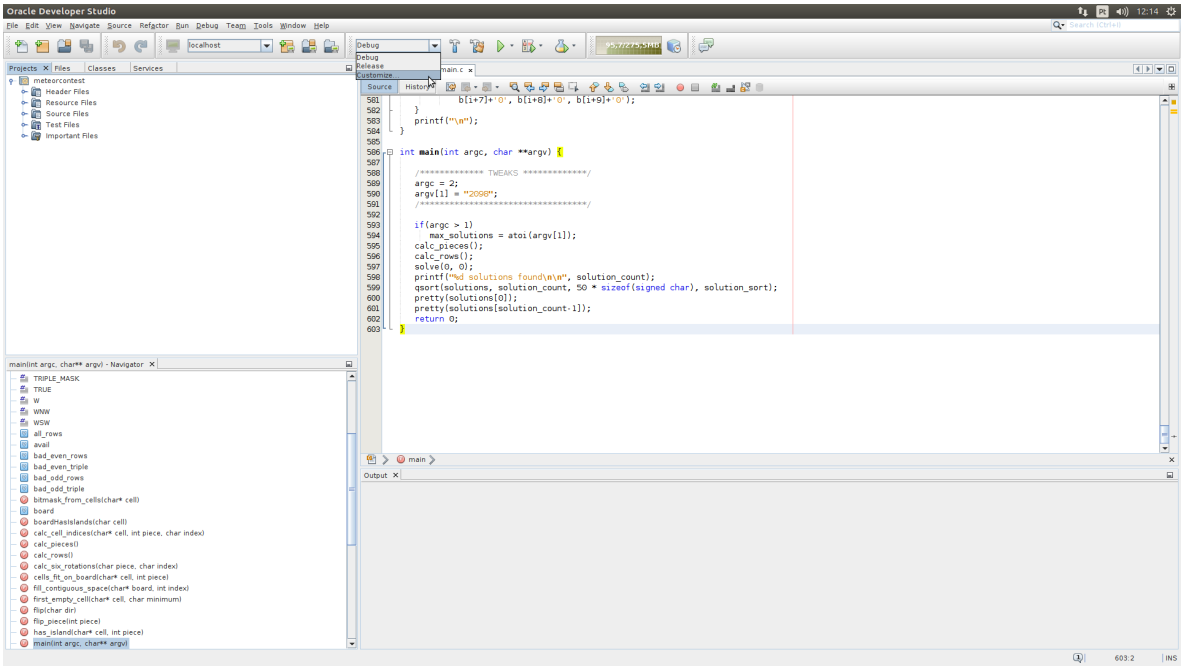


Figure 20.: Changing between profiles through Oracle Developer Studio.

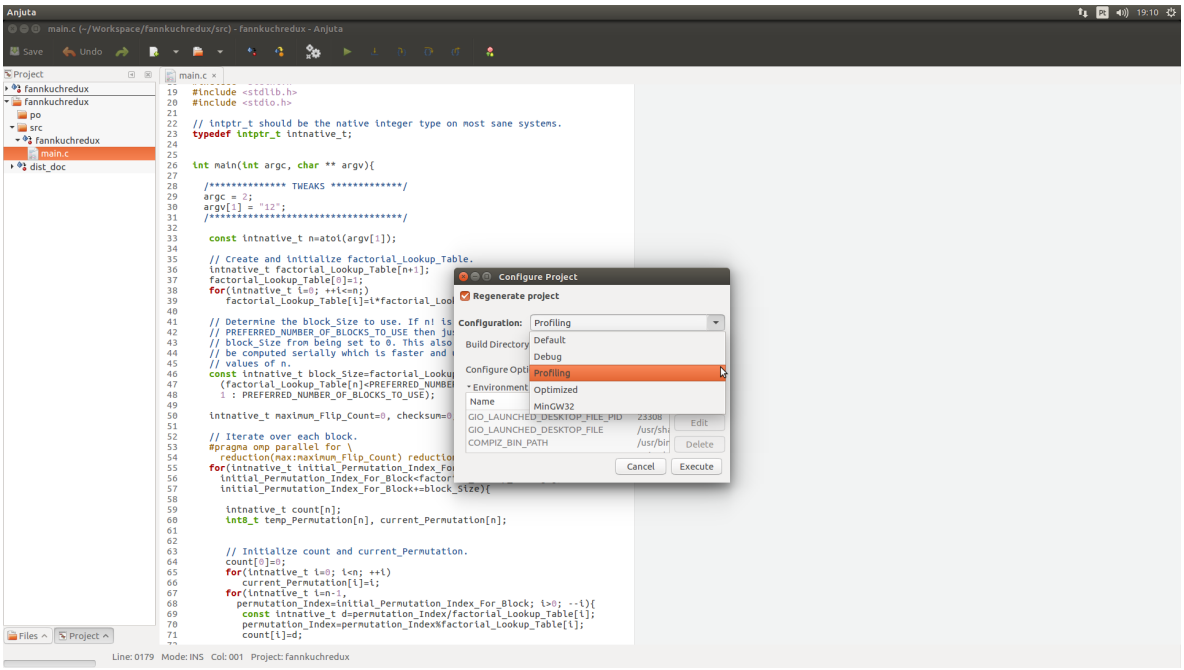


Figure 21.: Changing between profiles through Anjuta DevStudio.

There are 51 compilation profiles available in the total of the analyzed tools, of which 48 are non-empty and 29 are different from one another. Table 9 describes the profiles and respective parameters, being the first profile of each tool precisely which is used as default.

Tool ID	Tool Name	Profile Name	Profile Parameters
1	CMake	Debug	-g
		Release	-O3 -DNDEBUG
		RelwithDebInfo	-O2 -g -DNDEBUG
		MinSizeRel	-Os -DNDEBUG
2	qmake	Debug	-pipe -g -Wall -W -fPIC
		Profile	-pipe -O2 -g -Wall -W -fPIC
		Release	-pipe -O2 -Wall -W -fPIC
3	Qbs	Debug	-g -O0 -Wall -Wextra -pipe -fvisibility=default -fPIC
		Release	-O2 -Wall -Wextra -pipe -fvisibility=default -fPIC -DNDEBUG
4	NetBeans IDE	Debug	-g -MMD -MP -MF
		Release	-O2 -MMD -MP -MF
5	Code::Blocks	Default	(none)
6	CLion	Integrates CMake	
7	CodeLite	Debug	-g -O0 -Wall
		Release	-O2 -Wall -DNDEBUG
8	Eclipse CDT	Debug	-O0 -g3 -Wall -fmessage-length=0
		Release	-O3 -Wall -fmessage-length=0
9	KDevelop	Integrates CMake	
10	Geany	Default	-Wall
11	Anjuta DevStudio	Default	(none)
		Debug	-g -O0
		Profile	-g -pg
		Optimized	-O2
12	Qt Creator	Integrates CMake, qmake and Qbs	
13	DialogBlocks	Debug	-O0 -ggdb -Wall -Wno-write-strings
		Release	-O2 -Wall -Wno-write-strings
14	ZinjaI	Debug	-fshow-column -fno-diagnostics-show-caret -g2 -Wall -O0
		Release	-fshow-column -fno-diagnostics-show-caret -Wall -O2
15	GPS	Default	(none)
		SomeOpt	-O
		FullOpt	-O2
		FullAutoInli	-O3
16	Oracle Developer Studio	Debug	-g -MMD -MP -MF
		Release	-O2 -MMD -MP -MF
17	Sphere Engine	Default	-O2 -lm -fomit-frame-pointer
18	AWS Cloud9	Debug	-ggdb3 -std=c99
		Default	-std=c99

Table 9.: Tools, profiles and parameters analyzed.

Each examined tool gives a maximum of 4 different profiles per project, being the most usual case the presence of 2. The strong versatility of Qt Creator allows it to accumulate up to 9 distinct profiles in different projects through the integration of the 3 selected BATs.

There is clearly a tendency in the goal choices provided by the tools in question. From a global point of view, the three major categories in terms of the options main purpose are: profiles with no parameters, profiles that aim to provide the generated code with debugging/profiling information (that may or may not contain warnings and some optimizations) and profiles focused mainly on optimizations (which may vary in degree and purpose and even contain debug information). Another aspect that is quite clear is the tools choice in providing as a default profile one that has debugging characteristics and absence of optimization parameters.

In the collection obtained it is verified that for 11 profiles there is at least one other that is constituted by the same parameters. This repetition is due to several factors, in particular because the set is very wide (thus making repetitions more and more propitious), some profiles are fairly minimalist (providing only 1 parameter or even being empty), due to the integration of BATs into IDEs (inheriting also the same predefined configurations) and also because the design of some tools are strongly based on others (similarly replicating their base settings).

CLion, KDevelop, and Qt Creator are three of the many IDEs that delegate the management of build profiles to external tools. They create CMake-based C/C++ projects with the respective CMakeLists.txt template along all the necessary settings for a simple and functional application. As regards the QT Creator, this fact is even more significant because it has the ability to integrate an even larger set of tools (including qmake and Qbs).

For different reasons but with similar results, it is also possible to notice that both NetBeans and Oracle Developer Studio provide the user with exactly the same profiles and their compilation parameters. In this particular case it is due to the Oracle Developer Studio design being strongly based on NetBeans and therefore replicates most of its behaviors and configurations (Oracle team, 2018).

Some IDEs also provide the user with the ability to create their own profiles. Generally through iterative menus, drop-down lists and as well with the help of informative notes, the user can easily manage a set of options related with the profiles such as the name, warnings detail, architecture, standard C dialect, multithreading, optimization level, among others. In the Figures 22 and 23 are well demonstrated the mentioned options in two of the analyzed tools.

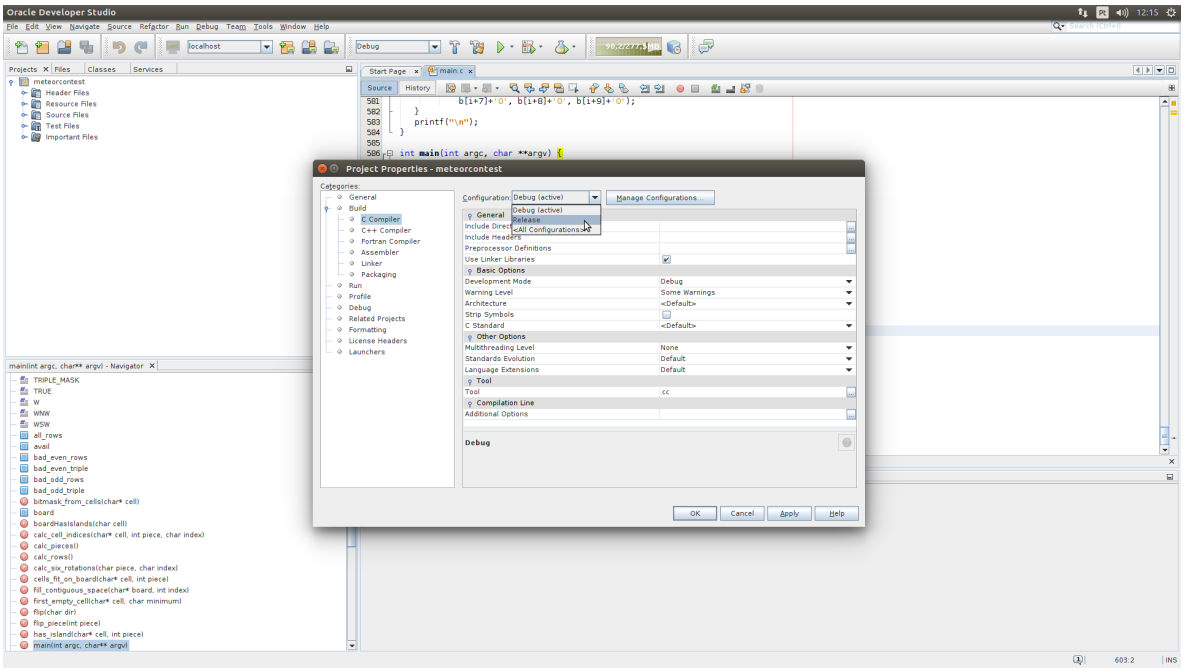


Figure 22.: Managing profiles through Oracle Developer Studio.

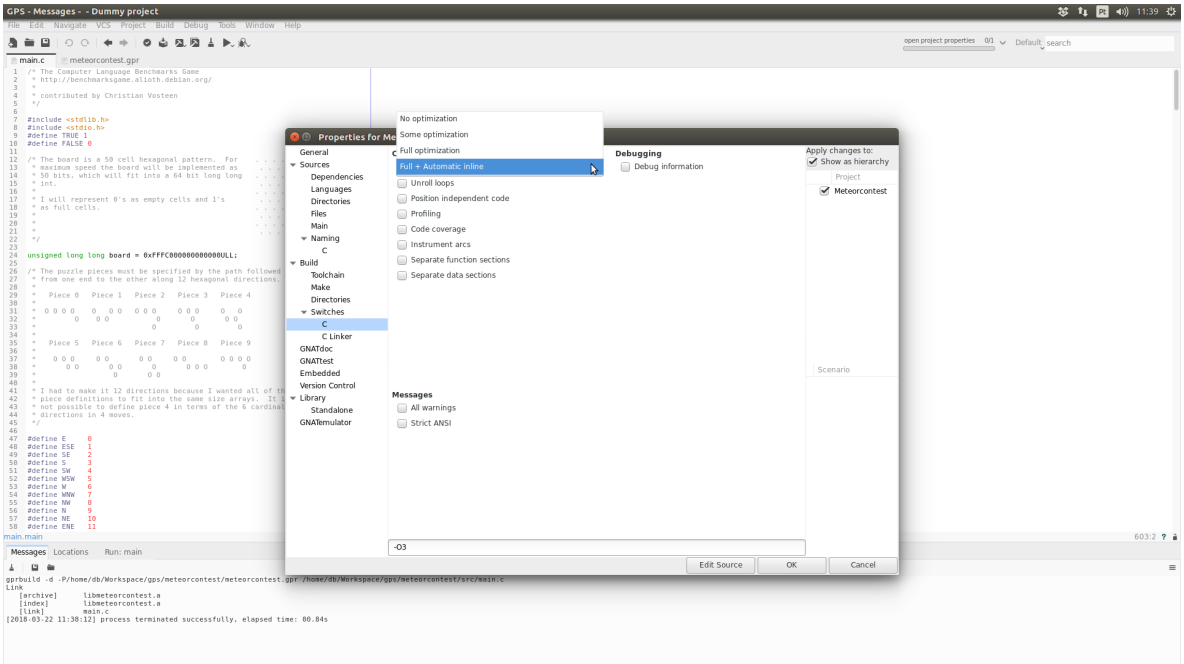


Figure 23.: Managing profiles through GPS.

From what could be observed in our research, it is possible that for other versions and types of IDEs, Operating Systems and machine architectures there are some differences

between the values indicated in table 9 and the values presented in such cases. For example, Anjuta DevStudio also has another profile called MinGW32 which was not considered for our study (because it is specific to Windows — among other reasons). It is also known that some tools (e.g. qmake) contains in their profiles the -m32 flag for compatibility issues and to generate code for 32-bit environment. Therefore, it is important to mention that the data set we obtained is quite broad, but still specific for the parameters of this study and respective machine used.

Profiles Parameters

Despite of the great diversity of options provided by GCC for C language, in practice it is clear that the analyzed tools only use options from a smaller set of categories. In fact, they only use options from 9 categories and do not mix more than 4 per profile. It is also verified some particular incidence on options focused on producing debug information, warnings management and code optimization.

Following will be made a brief contextualization of what each category consists of, along with a description of all the parameters obtained (GCC team, 2018b) (Saddler, 2018) (Von Hagen, 2011). All information given below are updated according to the version of GCC used in this study (7.2.0).

PREPROCESSOR OPTIONS

Options that control the C preprocessor that are executed in the first phase of the process, immediately prior to the actual compilation. In this stage continued lines and stripping comments are joined, lines of code beginning with # character are interpreted as preprocessor commands, among other tasks with preliminary nature. GCC allows the process to terminate immediately after this phase (generating the respective output) by using the -E option argument.

- **-MMD -MP -MF**
  Set of options related with preprocessing the output to a format adapted to makefiles. Instead of the usual result, the preprocessor generates one make rule suitable for describing the dependencies of the main source file. With the command that includes the three mentioned options, the dependency management of the source files with the makefile becomes more automated and allows it to react more properly. For example, with this mechanism it becomes possible that if a header file is changed all source files that include it are recompiled.
- **-D*name***
  The -D option allows to set the *name* parameter as a macro with a value of 1. NDEBUG is a macro that when set allows to turn off asserts as mandated by the

C standard library (assert ()). In turn, active asserts allow diagnostic information to be written to the standard error file.

C DIALECT OPTIONS

Control parameters related to which C dialect (or derived languages) the compiler should accept.

- **-std=c99**
  Option that allows to configure which language standard to be chosen by the compiler from more than 30 possible. If they were compatible, it can even be chosen several base standards (and the GNU dialects of those standards). Programs that do not respect the specified parameters (and possibly some strict-conforming) are considered invalid and rejected by the compiler. If no value is specified, the dialect considered is the gnu11.

  In 1999 the new edition of the ISO C standard was published (commonly known as C99), improving some aspects of the language and introducing some new features such as inline functions, flexible array members, designated initializers, several new data types (e.g. long long int), etc.

DIAGNOSTIC MESSAGE FORMATTING OPTIONS

Options that allow to format the appearance of the diagnostic messages according to the output device's aspect (for example its height, width, etc.). By default there is no care taken by the GCC in this regard, leaving to the user the customization of aspects such as how often source location information should be registered or how many characters should appear per line.

- **-fmessage-length=*n***
  Option that allows to format the error messages according with the maximum *n* characters per line. By default, the value is 0 and no line-wrapping is done, being made each error message available per line.
- **-fno-diagnostics-show-caret**
  Each diagnostic message produced includes, besides the original source line, also a caret '^' indicating the respective column. With this flag is possible to suppress this piece of information. When used in conjunction with the previously displayed flag, the source line is truncated to its *n* characters.
- **-fshow-column**
  Flag that is defined by default and controls whether or not GCC prints the column number of a diagnostic message.

WARNING OPTIONS

Compiler warnings are diagnostic messages that describe detected anomalies in which

present some risk to result in an error. After some assumptions, the compilation process proceeds and the special situation detected is reported to the user through a message (not with the reason that produced it, but rather with the low level anomaly detected by the compiler). Although these messages are disabled by default and sometimes can be very inconvenient and difficult for the user to discover the message cause, they should not be ignored at all but treated with due care in order to increase the confidence of the code produced. Options in this category allows to control the number and types of warnings produced by GCC, or, in other words, how picky should the compiler be.

- **-Wall**
  Activates all the warnings regarding to constructions types that may raise doubts (including in conjunction with macros) or approaches that should be avoided, being at the same time easy to correct in order to prevent the warnings in question. In total, there are 50 options that can be activated/deactivated depending of the source code and dialect used.
- **-W, -Wextra**
  Option that activates some extra warning flags that are not included in the -Wall option and that has the same purpose: to help ensure the underlying code is secure. Do not confuse with the -w flag that inhibits all warning messages. The nomenclature used in the most recent compiler versions is -Wextra and this option should be preferably used since it is more descriptive (although the old one is still supported). Depending on the type of source code, dialect used and if it is applied in conjunction with other parameters, this option enables/disables up to 15 flags and provides 6 new warning messages (mainly for C++).
- **-Wno-write-strings**
  Suppresses warnings regarding attempts to copy or write to string constants that may appear dangerous or sloppy. Option selected by default by the compiler and its negation (-Wwrite-strings) is not included either -Wall or -Wextra.

DEBUGGING OPTIONS

Options to produce debuggable code.

Due to the multiple transformations applied to the source code throughout the compilation process, some machine instructions cannot effectively be mapped onto a specific source statement. In particular, due to the various optimizations when compiling, sometimes the program behavior becomes different from what would be expected when looking only at the source code. Among the many possible surprises it can be verified that, for example, some initially declared variables have been removed, some

instructions are not executed (due to constant results), changes in the control flow (e.g. giving the impression that the program is running in the opposite direction), etc.

In order to facilitate the work during the development phase, GCC provides options that allow the generated code to have useful information for a better analysis of the same and even to emit extra information to being used by a debugger tool. If no level of optimization is selected beyond the default, it is recommended to use the -Og for a better debugging experience option.

- **-g**
  Option that allows the addition of debugging information to the generated code in the operating system's native format (e.g. stabs, XCOFF, DWARF or COFF). This extra information can also be used by GDB[82] (The GNU Project Debugger). This option has several levels of information, being by default equivalent to level 2. It is generally observed that the greater the selected level, the greater the size of the executable produced.

  Unlike most C compilers, GCC allows to use this option with some optimization suits (e.g. -O/-O1 or -Og) making the process of debugging in optimized code more simplified.

- **-g*level***
  Possibility of producing debugging information, being its type and quantity defined according to the specified *level*.

  - **-g0**
    Predefined option chosen by GCC in which no type of debug information is produced.

  - **-g1**
    It produces minimal information such as descriptions of functions and external variables, including details on local variables and line numbers. Usually used for situations where it is not planned to debug the code, but still want to safeguard the need of making backtraces in some parts of the program.

  - **-g2**
    Default debug information (equivalent to -g). Tells to the compiler to store symbol table information in the executable relative to symbols of the source code such as their names, type info, files and line numbers where they came from, among others.

  - **-g3**
    Maximal debug level that includes extra information such as all the macro

82 https://www.gnu.org/software/gdb/

definitions present in the source code. When used in some debuggers, it is also possible to get macro expansions.

- **-ggdb**
  Option that provides debugging information in the most expressive format available for specific use on GDB tool (including GDB extensions if at all possible).
- **ggdb*level***
  Analogous to g*level*.

PROGRAM INSTRUMENTATION OPTIONS

Profiling is one of the most important aspect of software programming and raises some of the biggest challenges in large-scale projects. It allows to collect profiling statistics for code coverage analysis, obtain performance gains (e.g. identifying hotspots, dead code and bottlenecks) and even bugs detection (e.g. invalid pointer dereferences or out-of-bounds array accesses).

GCC supports a range of options that add to the code several types of run-time instrumentation that allows tracing or function-level instrumentation for debug or program analysis purposes.

- **-pg**
  Gprof[83] is a tool used to profile software, namely to analyze the execution time of functions. This option allows to generate inside the code extra suitable information to be used by gprof program. For best results, is recommended using it in both phases of compilation and linking.

OPTIMIZATION OPTIONS

Options that allow to control various types of optimizations related to the performance of the compilation process and the subsequent generated code, such as runtime, compile time, code size, memory usage, amount and type of debug/profile information, among others.

By default, the compiler's goal is primarily to reduce the cost of the process and produce the expected debugging results (-Oo). However, there are flags (either individually or in sets) that allow to change the compiler main purpose to something more sophisticated. At the expense of compilation time and according to the knowledge that the compiler has about the program, transformations are applied to generic and not optimized code making it into something tailor-made to the program and system.

As the level of optimization increases, the compiler will attempt to produce better performing code. However, that level should always be seen as an indicator to which

83 https://sourceware.org/binutils/docs/gprof/

heuristics within the compiler's optimization engine will try to achieve for an average system and never as an absolute guarantee of the best code. Besides that, it is also up to the user to control the trade-off that exists between the various aspects inherent to the software itself such as significant reductions in execution time may cause the code to lose debugging information or increase its size, among other variables.

- **-fomit-frame-pointer**
  Machine-independent flag that omits the frame pointer in register for functions that do not need it or for targets that do not always use it in standard calling sequence. It allows some functions to obtain an extra register available and also guarantees reductions in terms of code size and the execution path by removing the instructions needed for the management of frame pointers (save, set up and restore). This option is included in all optimization levels other than default.
- **Optimization Levels**
  GCC provides optimization suites that allow the user to control various aspects of the compilation process and resulting code. The seven sets that the tool presents by default are: O0, O1, O2, O3, Os, Og, and -Ofast. If several of these options are indicated as compilation parameters, only the last one will be considered effective.
  - **-O0**
    Level automatically selected by the compiler, where optimizations are explicitly disabled for the executable produced. It focuses mainly on reducing the compilation time and returning the expected debugging information. Option not recommended (except for debugging purposes) because it only operates dead code elimination and other fairly basic transformations.
  - **-O, -O1**
    Most basic optimization level that can activate up to 43 flags.

    With the cost of increasing the compilation time and its consumed memory (mainly for large function), the compiler attempts to produce code that runs faster and has a smaller size. A fairly balanced baseline level that performs a good job on the generated code without major resource detriment during the compilation, while still providing the possibility of debugging.
  - **-O2**
    Optimization level where are applied practically all optimizations that do not involve a space-speed trade-off. The compiler attempts to improve code performance without neglecting too much its size and the required compile time.

It is, therefore, an optimization level superior than -01, where can be activated all the flags which are contained in it and even more 47 new ones (counting a total of 90). In general terms it turns out to be a level of optimization quite effective for most situations, unless of some special need of the code or machine.

- **-O3**
  With the possibility of applying up to 102 optimization flags (12 new plus the remaining ones contained in -O2), this option manifests itself as the highest level of optimization possible which does not disregard strict standards compliance. The compiler tries to get improvements by using some heuristics and intensive transformations such as loop unrolling, function inlining (even if they are not declared inline), avoid false dependencies in scheduled code, among others, but which continue to be valid for all standard-compliant programs. However, all the sophistication it presents has a great cost in terms of compile time, memory usage, increasing the difficulty of debugging the code (it becomes even practically impossible) and sometimes also the size of the code generated.

- **-Os**
  Level designed to optimize for code size. It encompasses all the optimizations present in -O2 that do not affect the generated code size, counting a total of 82 possible flags. This option is widely considered in applications and machines with large limitations of disk storage and/or CPUs with small cache sizes.

OVERALL OPTIONS

Options that allow to control the type of output generated by the GCC: an executable, object files, assembler files, or preprocessed source. It gives the user greater process control, making possible for example to execute each phase independently, re-use the previous output or start the process from any stage. The suffix from the name of any input file received by GCC determines which type of compilation phase will be performed.

- **-pipe**
  This option tells the compiler to preferentially use pipes for communication during the build phases instead of the predefined temporary files. Although it has no impact on the generated code, it makes the compilation process faster and cleaner. However, it naturally causes an increase in the memory required and for more limited systems its use may even be impracticable.

OPTIONS FOR CODE GENERATION CONVENTIONS

Machine-independent options that allow controlling various aspects related to the interface conventions used in code generation.

- **-fPIC**
  Generate position-independent code (PIC) suitable for dynamic linking, which means that code can be loaded at any particular virtual memory address at run-time. In practice it means that the compiler adds an extra level of indirection for accessing static/global variables and functions, being especially useful to avoid any imposed limit on the size of the global offset table.

  For some architectures such as AArch64, m68k, PowerPC and SPARC, among others, its use has a lot of impact and sometimes even becomes a mandatory requirement. For the remaining supported target machines (as the one in this study) it remains a useful option due to position-dependent memory references can't be shared by different processes and thus are copied.

- **-fvisibility=*default***
  Option that allows to change the ELF image symbol visibility, being able to have implications in the improvement of the linking and load times of shared object libraries, prevent symbol clashes, produce more optimized code and provide near-perfect API export.

  It is possible to select between *default*, *internal*, *hidden* and *protected*, being the first one as the name itself indicates the option predefined and which deals to make every symbol public.

OPTIONS FOR LINKING

Linking is the designation of the last step of the compilation process in which, briefly, proceed to the agglomeration of several object files into a single executable binary. With options of this category it is possible to use GCC to configure that process as well as to indicate all the files that are intended to be part of the final result. This phase is not performed if any of the -E, -c or -S parameters were previously used.

- **-l*lib***
  Option that instructs the linker to look in a standard list of directories for the library named *lib* when linking. Most libraries are a collection of precompiled object files that usually provide program with system functions. It is also possible to add other search directories by including the -L flag. Through searching for files with this name (which are usually files with .a or .o extension), the linker tries to match symbols that have been referenced but which are still undefined at the moment.

- **-lm**
  Library present in the C standard library which implements basic but essential mathematical functions like sqrt, sin, cos, log, etc.

Also some of the programs add an extra parameter to the compilation process in addition to those applied by the analyzed tools. Being mostly related to aspects of linking libraries, in total there are seven programs under study that require an auxiliary option so that it is possible to compile them.

- **-pthread**
  Option that adds the macros definition needed to use the POSIX threads library which allows, for example, a program to control multiple different flows of work that overlap in time. Thread-ring and chameneos-redux are the programs that use this feature during their executions.

- **-lm**
  N-body and spectral-norm require the inclusion of the header math.h[84], which is where precisely the most mathematical functions are declared.

- **-lapr-1**
  The Apache Portable Runtime (APR) contains a set of APIs that map to the underlying operating system and provide interface to platform-specific implementations. The binary-trees program requires its inclusion, in particular the header that provides APR memory allocation[85].

- **-lgmp**
  Library that has a wide set of functions that allow to deal with arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers. Pidigits needs GMP[86] library so that it may be able to, for example, use integers that can grow dynamically to the required precision and dynamically allocate memory for accommodating extra bits of precision as and when needed.

- **-lpcre**
  The Perl-compatible regular expression (PCRE[87]) is a library that allows to handle with the regular expression pattern matching using the same syntax and semantics as Perl 5. Regex-redux makes use of this functionality by including the header file pcre.h.

84 http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/math.h.html
85 https://apr.apache.org/docs/apr/1.6/apr__pools_8h.html
86 https://gmplib.org/
87 https://www.pcre.org/

Parameters Analysis

Taking into account the categorization indicated by GCC as well as the options contained in the gathered profiles, it is possible to briefly and less rigorously group both aspects into four main groups according to their applicability: control of the compilation process, management of diagnostic messages, availability of debug/profile information and optimization of generated code. Below is presented the referred parameters categorization, together with their total occurrences.

1. **Compilation management options**
   Configuration of preferences involving the compilation process, definition of macros and dialects, management of dependencies.
   - -MMD -MP -MF (4)
   - -std=c99 (1)
   - -DNDEBUG (15)
   - -pipe (10)
   - -fPIC (10)
   - -fvisibility=default (4)
   - -lm (1)
2. **Diagnostic messages options**
   Management of aspects related to indications given to the user about anomalies detected during the compilation process.
   - -fmessage-length=0 (2)
   - -fno-diagnostics-show-caret (2)
   - -fshow-column (2)
   - -W, -Wextra (10)
   - -Wall (19)
   - -Wno-write-strings (2)
3. **Debug and Profile options**
   Specifications to produce debuggable code and collecting profiling statistics.
   - -g, -g2 (21)
   - -g3 (1)
   - -ggdb (1)
   - -ggdb3 (1)

- -pg (1)

4. **Optimization options**
   Parameters to enable/disable code optimizations.
   - -fomit-frame-pointer (1)
   - -Oo (7)
   - -O (1)
   - -O2 (18)
   - -O3 (6)
   - -Os (4)

In total, 144 parameters (28 distinct) are used in the profiles obtained in this study.

It is important to note that some flags may have influence on several of the aspects previously presented and can even change of category depending on the value that they have. For those cases where it is found that changing the flag value allows obtaining a different effect, then their inclusion in another category will prove to be more appropriate. For example, the -fvisibility option was included in the first category because its value is *default* and not one of the remaining possible (*internal*, *hidden*, *protected*) otherwise it could be considered a flag with an optimization effect.

Of the 51 compile profiles obtained, 34 have more than 1 parameter and mostly they embrace different aspects and categories. In fact, considering only the distinct profiles, this factor becomes even more significant covering 76% of the cases (22 out of 29). Consequently, in addition to the individual meaning of each option and its respective effect on the compilation process, it is also interesting to think in each compilation profile as a whole taking into account the purpose of the flags that compose it. With that perspective in mind, all the profiles obtained were considered and categorized into 9 different groups:

1. **Empty** (3)
   Profiles without any kind of parameter.
   - Anjuta DevStudio - Default
   - Code::Blocks - Default
   - GPS - Default

2. **Warning/Dialected oriented** (2)
   Profiles with only diagnostic message flags.
   - Geany - Default: ”-Wall”

- AWS Cloud9 - Default: ”-std=c99”

3. **Debugging/Profiling without warnings** (8)
   Adding debugging/profiling information without considering diagnostic messages.
   - Anjuta DevStudio - Debug: ”-g -O0”
   - Anjuta DevStudio - Profile: ”-g -pg”
   - CMake - Debug: ”-g”
   - CLion - Debug: ”-g”
   - KDevelop - Debug: ”-g”
   - QTCmake - Debug: ”-g”
   - NetBeans IDE - Debug: ”-g”
   - Oracle Developer Studio - Debug: ”-g”

4. **Debugging with warnings** (9)
   Profiles that provide with both diagnostic messages and debugging/profiling information.
   - Eclipse CDT - Debug: ”-O0 -g3 -Wall -fmessage-length=0”
   - CodeLite - Debug: ”-g -O0 -Wall”
   - Qmake - Debug: ”-pipe -g -Wall -W -fPIC”
   - QTQmake - Debug: ”-pipe -g -Wall -W -fPIC”
   - ZinjaI - Debug: ”-fshow-column -fno-diagnostics-show-caret -g2 -Wall -O0”
   - Qbs - Debug: ”-g -O0 -Wall -Wextra -pipe -fvisibility=default -fPIC”
   - QTQbs - Debug: ”-g -O0 -Wall -Wextra -pipe -fvisibility=default -fPIC”
   - DialogBlocks - Debug: ”-O0 -ggdb -Wall -Wno-write-strings”
   - AWS Cloud9 - Debug: ”-ggdb3 -std=c99”

5. **Low Level Optimization** (1)
   Optimization with the most basic level.
   - GPS - SomeOpt: ”-O”

6. **Optimization with debug information** (6)
   Profiles with code optimization and also debugging information (in two cases also with warnings).
   - CMake - RelwithDebInfo: ”-O2 -g -DNDEBUG”
   - CLion - RelwithDebInfo: ”-O2 -g -DNDEBUG”

- KDevelop - RelwithDebInfo: "-O2 -g -DNDEBUG"
- QTCmake - RelwithDebInfo: "-O2 -g -DNDEBUG"
- Qmake - Profile: "-pipe -O2 -g -Wall -W -fPIC"
- QTQmake - Profile: "-pipe -O2 -g -Wall -W -fPIC"

7. **Optimization Recommended Level** (12)
   Optimization with the most recommended level in generic terms.
   - CodeLite - Release: "-O2 -Wall -DNDEBUG"
   - Anjuta DevStudio - Optimized: "-O2"
   - DialogBlocks - Release: "-O2 -Wall -Wno-write-strings"
   - Qmake - Release: "-pipe -O2 -Wall -W -fPIC"
   - QTQmake - Release: "-pipe -O2 -Wall -W -fPIC"
   - GPS - FullOpt: "-O2"
   - Qbs - Release: "-O2 -Wall -Wextra -pipe -fvisibility=default -fPIC -DNDEBUG"
   - QTQbs - Release: "-O2 -Wall -Wextra -pipe -fvisibility=default -fPIC -DNDEBUG"
   - NetBeans IDE - Release: "-O2"
   - Oracle Developer Studio - Release: "-O2"
   - Sphere Engine - Default: "-O2 -lm -fomit-frame-pointer"
   - ZinjaI - Release: "-fshow-column -fno-diagnostics-show-caret -Wall -O2"

8. **High Level Optimization** (6)
   Profiles with the highest optimization level (without breaking strict standards compliance).
   - CMake - Release: "-O3 -DNDEBUG"
   - CLion - Release: "-O3 -DNDEBUG"
   - KDevelop - Release: "-O3 -DNDEBUG"
   - QTCmake - Release: "-O3 -DNDEBUG"
   - Eclipse CDT - Release: "-O3 -Wall -fmessage-length=0"
   - GPS - FullAutoInline: "-O3"

9. **Code Size Optimization** (4)
   Optimizations that emphasize the production of executable with reduced size.
   - CMake - MinSizeRel: "-Os -DNDEBUG"
   - CLion - MinSizeRel: "-Os -DNDEBUG"

- KDevelop - MinSizeRel: "-Os -DNDEBUG"
- QTCmake - MinSizeRel: "-Os -DNDEBUG"

As expected, the main focus from the gathered profiles it rests mainly in two aspects: debugging and optimization. Considering the 51 profiles, 23 use debugging and 30 optimization options, 6 select parameters from both categories and only 5 choose not to include any of those types. Excluding Code::Blocks (which only provides an empty profile), in fact only Geany, GPS and Sphere Engine do not use any debugging option in their profiles and only 2 tools do not use any optimization parameter: Geany and Cloud9.

Naturally, debugging options are mostly associated with the absence of code optimization levels, since as previously mentioned greatly increases their effectiveness.

Most of the profiles opted for the default debug information level (2), with the exception of 2 cases that opted for the maximum level (profiles from Eclipse and AWS Cloud9). There is, however, some balance between the number of profiles with debugging/profiling information that opt-in or not for also issuing diagnostic messages.

In fact, roughly half of the tools do not consider relevant the use of warnings, completely ignoring parameters of that category and the respective inherent benefits. Given their importance, the option of not including any parameters in at least in the profiles related with initial states of software development (e.g. default, debug) reveals some strangeness. Although sometimes the returned messages are not very clear, we consider however that they contain information relevant to the user and should not be purely discarded.

Qbs and qmake prove to be the 2 most complete tools at this level using in all profiles 2 parameters that can activate up to 65 diagnostic messages. This also extends to Qt Creator if the user choose to work with one of the mentioned tools.

For profiles that have suites that operate source code optimizations, most include the one that is considered most effective in generic terms. 18 profiles have selected -02, 6 choose to use -O3 trying to get even more improvements in the generated code (despite the inherent disadvantages), 4 provide the possibility to optimize for code size and only 1 from GPS chooses the one that is the most basic level and balanced in terms of trade-off compilation-execution (-O1).

In addition to the -Os level, no profile chooses to use any of the other specific optimization levels: -Og and -Ofast.

-Og has the ability to provide a superior debugging experience along with a fast compilation and a reasonable level of runtime performance. It presents a better development experience than the -O0 level and turns out to be a great choice for the standard edit-

compile-debug cycle. This option should be more considered, especially in profiles that do not opt for any of the optimization suits beyond the default (GCC team, 2018b).

-Ofast allows a higher level of optimization than the one obtained with the suit -O3, but disregards strict standards compliance. Although its use is not recommended for most cases, it can however, for more specific ones, allow considerable gains in interesting aspects (such as the program execution time) and therefore not should be a totally discarded option (GCC team, 2018b).

No tool chooses to go any further than optimization suites and for example add/remove some individual flag. Although the Sphere Engine does indeed refer to an extra optimization flag, it is already included in the -O1 suite and -O2 respectively (level that is also considered in the respective profile).

Interestingly, three tools indicate as the default profile an empty list of parameters, letting the process unfold completely according to the compiler's default options. Geany decides for a bit more in its unique profile, opting to add one flag related to the listing of warnings to the user.

Also curious is the option taken by Cloud9 that, unlike the other tools, chooses to change the dialect of C to be used by the compiler. By default, GCC uses the gnu11 which is a superset of C11 that is the current standard for C programming language released in 2011. However, Cloud9 by default prefers to tell the compiler to use the C99, which is a version prior than the current standard and could lead to problems such as code compatibility. Regardless of the reasons associated with this decision, we consider that the choice of the ideal dialect should always depend on the specificity of the project and the user's own preferences, as well as that there should always be some care when choosing a choice other than the community standard.

It is possible to notice that Sphere Engine tells GCC to link the math library (with -lm) for any program. It is an unsophisticated approach since this should only be necessary if it were found that the code in question really needs that library. It is true that due to referring to a static library it will only be included in the final code if in fact it is really necessary, because if there is no undefined symbol during the linking process they will be ignored. However, linking unused libraries presents other drawbacks such as increasing the static linking time (because symbol resolution time in more libraries) and there is an increase in the size of files generated during the process (for example the ELF file).

We also verified the existence of some parameters in the presented profiles that did not need to be explicitly declared, because in the context in which they are inserted they are already part of the set of options that the compiler takes. The most frequently repeated case is the -O0, but there are also others previously mentioned such as -Wno-write-strings or

-fshow-column. Another type of redundancy observed is the specification of the parameter value that is equally irrelevant to the compiler because it is precisely its default value (e.g. -g2, -fvisibility=*default*).

5.5.3 *Measurement Process*

After completing all the research work, and keeping in mind the objectives intended for the study, it is now mandatory to orchestrate a methodology that includes in a functional and efficient way all the elements gathered during the previously described stages: testing platform with Linux system and Intel processor, framework that uses C language and RAPL to software measurement, 12 C benchmarks compiled with GCC and using parameters obtained from 51 compilation profiles (with a total of 144 parameters) present in 18 tools. In addition to connecting elements, it is also important to use that methodology to verify which aspects are relevant or not to analyze and also if the generation and processing of results is in accordance with the intended analysis, not becoming for example a cluster of intractable values or a process too expensive.

Selection and Filtering of Compilation Profiles

Due to incompatibility reasons between the programs to be measured and the options present in compilation profiles, the -std=c99 parameter was discarded from the following analysis. As previously mentioned, this option allows to configure which language dialect to be chosen by the compiler. However, in practice it is verified that not all programs are compatible with that parameter since they were written considering another dialect such as gnu11 (which is the current standard GCC dialect). This aspect becomes very relevant because it does not allow the successful compilation of some of the present benchmarks, thus making it impossible to execute them and the remaining analyzes intended. Therefore, the -std=c99 parameter was taken from both the AWS Cloud9 profiles for the purposes of the intended measurements.

Due to irrelevance reasons, the parameters segment -MMD -MP -MF present in the Oracle Developer Studio and NetBeans IDE tools profiles was also disregarded from the performed measurements. These options have as main purpose the preprocessing of output to a format adapted to makefiles, aiding the management of dependencies in a phase prior to the actual compilation itself. Since its role is limited to this type of tasks and has no influence on the generated code, it is therefore irrelevant to its execution and consequently to the desired measurements.

In order to make the measurement process more effective and reduce the redundancy of results, a filtering was performed on the compilation profiles. As previously mentioned, there are some repetitions among the 51 profiles obtained in terms of their content as a whole. Naturally, this leads to identical compilations and executables for such cases, which consequently causes redundancy in the measurements performed. In addition to the 22 repeated profiles already mentioned before, were also disregarded another 3 new cases that, due to the removal of parameters performed in the previous paragraphs, they also become equal to others already existing.

Having said all this, a group of 26 compilation profiles was considered for the purpose of measuring the energy consumption and the time of execution of the collected programs, which results are identical for the remaining 25 elements that make up the total set of profiles obtained. Table 10 describes how the profiles are grouped in terms of the stated selection and filtering aspects.

Identifier	Tool - Profile Name	Profile Parameters
1	Anjuta DevStudio - Default	(none)
	Code::Blocks - Default	
	GPS - Default	
	AWS Cloud9 - Default	
2	Geany - Default	-Wall
3	Anjuta DevStudio - Debug	-g -O0
4	Anjuta DevStudio - Profile	-g -pg
5	AWS Cloud9 - Debug	-ggdb3
6	CMake - Debug	-g
	CLion - Debug	
	KDevelop - Debug	
	QT - CmakeDebug	
	NetBeans IDE - Debug	
	Oracle Developer Studio - Debug	
7	Eclipse CDT - Debug	-O0 -g3 -Wall -fmessage-length=0
8	CodeLite - Debug	-g -O0 -Wall
9	Qmake - Debug	-pipe -g -Wall -W -fPIC
	QT - QmakeDebug	
10	ZinjaI - Debug	-fshow-column -fno-diagnostics-show-caret -g2 -Wall -O0
11	Qbs - Debug	-g -O0 -Wall -Wextra -pipe -fvisibility=default -fPIC
	QT - QbsDebug	
12	DialogBlocks - Debug	-O0 -ggdb -Wall -Wno-write-strings
13	GPS - SomeOpt	-O
14	CMake - RelwithDebInfo	-O2 -g -DNDEBUG
	CLion - RelwithDebInfo	
	KDevelop - RelwithDebInfo	
	QT - CmakeRelwithDebInfo	
15	Qmake - Profile	-pipe -O2 -g -Wall -W -fPIC
	QT - QmakeProfile	
16	CodeLite - Release	-O2 -Wall -DNDEBUG
17	DialogBlocks - Release	-O2 -Wall -Wno-write-strings
18	Qmake - Release	-pipe -O2 -Wall -W -fPIC
	QT - QmakeRelease	
19	Anjuta DevStudio - Optimized	-O2
	GPS - FullOpt	
	NetBeans IDE - Release	
	Oracle Developer Studio - Release	
20	Qbs - Release	-O2 -Wall -Wextra -pipe -fvisibility=default -fPIC -DNDEBUG
	QT - QbsRelease	
21	Sphere Engine - Default	-O2 -lm -fomit-frame-pointer
22	ZinjaI - Release	-fshow-column -fno-diagnostics-show-caret -Wall -O2
23	CMake - Release	-O3 -DNDEBUG
	CLion - Release	
	KDevelop - Release	
	QT - CmakeRelease	
24	Eclipse CDT - Release	-O3 -Wall -fmessage-length=0
25	GPS - FullAutoInline	-O3
26	CMake - MinSizeRel	-Os -DNDEBUG
	CLion - MinSizeRel	
	KDevelop - MinSizeRel	
	QT - CmakeMinSizeRel	

Table 10.: Measured Profiles.

Benchmarking and Output Processing

Briefly, the methodology that encompasses the process of programs measurement and output handling can be described through the following steps:

1. Compile the measurement framework;
2. Execute of the warm-up program;
3. Select the desired program;
4. Select the compilation profile to apply;
5. Adjust the respective makefile according to the options taken in 3 and 4;
6. Compile the program as stipulated in the makefile;
7. Execute 50 times the measuring tool for the obtained executable;
8. Processing of the output generated by the measuring tool:
    a) Obtain the execution time and energy consumption values for each analyzed case;
    b) Ignore the 10 highest and lowest values;
    c) Calculate the average for the remaining 30 values;
    d) Generate a table and plot with the results in an HTML page.
9. Repeat steps 4 to 8 for the remaining profiles;
10. Repeat steps 3 to 9 for the remaining programs.

Considering the type of programs to be analyzed, as well as the results obtained in the previous study and in some preliminary tests), we chose to discard the values measured for the GPU (since they are approximately 0 for all cases) and perform the measurements in only one CPU core.

In addition to the mentioned steps, some preventive measures have been taken in order to reduce disturbances related to the study machine and remaining elements (e.g. pre-loading of data, memory heating, network or other programs interference, etc.) and to ensuring the conformity of the process and its results. The methodology presented was repeated more than a dozen times in the target machine as a superuser. The measurement process for a given program was always performed uninterrupted for all its profiles and never in an isolated way, but rather in sessions that lasted between 8h to 36h. Before and during all sessions the machine was always in a state of similar operability, without any human iteration,

any connected peripherals, any other relevant program running concurrently and without access to any network. Also in order to promote the operation similarity, the measurement framework was executed in a generic program (during approximately 10 minutes) at the beginning of each session, functioning as a validation and warm-up step for the whole process and its components.

In conjunction with the design of the measurement process, a bash script was also created which allows through its invocation to automate all the steps of referred measurement process. In concrete, there is a file called *read.sh* that has the main algorithm who performs tasks such as some preliminary checks, directories management, selection of elements to analyze, change of makefiles options, programs compilation according to the parameters defined, compilation and invocation of the measurement framework and finally the processing of the respective output. There is also an auxiliary file called *sources.sh* where all the elements to be analyzed, whether programs or compilation profiles, are declared in the form of variables of arrays and associative arrays.

This tool also manages the creation of a directory tree that provides not only a better process organization but also increases its automation capacity, allowing, for example, that all programs contained in a directory can be analyzed automatically or that the generated output is forwarded to a place more appropriate for its treatment. It should be noted that the script still allows other ways of operating depending on the arguments passed at its invocation, but that were not considered for this methodology in particular.

The output processing (referred in item 8 above) was performed entirely through a Perl script. In general terms, its behavior consists of in a first stage parsing and storing in some data structures all the relevant information of the programs and their values obtained. Then in sorting each of the arrays of the respective 3 factors intended for analysis (execution time and energy consumptions of CPU and memory), remove the 10 highest and lowest values (in order to discard possible errors and measurements with disturbances) and calculate the mean of the resulting 30 values for each factor. After organize all the necessary information, with the assistance of 2 Perl libraries (Chart::Gnuplot and HTML::Table), the relevant data is illustrated through charts, tables, HTML pages and rankings.

Due to the great versatility of the developed measurement framework, there would indeed be the possibility of adopting other more efficient workflow for this particular study. We opted for the solution described in the presented methodology that passes through the successive invocation of the tool after the external compilation of the program under analysis in order to increase the independence between measurements, to facilitate the output

processing stage and to reduce the burden of the framework (delegating management tasks to the bash script that invoking it).

5.6 DISCUSSION OF RESULTS

In this section, the final results are exposed and discussed using the already mentioned methodology. The analysis focus mainly on the execution time, CPU and memory RAM energy consumption (individually and together) and the ratio between both of these factors (energy consumption/time). It is also intended that, based on all the addressed perspectives, the analysis relates to all the elements covered, more specifically, from the point of view of the studied programs, programs-tools, tools-profiles, profiles-parameters and of the individual compilation parameters obtained. Objectively, the approach will follow a path where initially it will focus on a more general and global perspective in terms of the values obtained in the performed measurements, and throughout the section, will gradually adjust to the elements with which they interact and the specifics that motivate them.

To make the analysis cleaner and more efficient, the previously collected information was treated and presented in several relevant types. These results are produced based on different types of scopes (e.g. execution time and energy consumption) and analyzed elements (e.g. programs, tools, profiles and parameters). That information already takes into account all logistic specifications and obstacles (for example using different profile categories or analyze different decimal place values).

Regardless of the target element represented, for all instances the data obtained regarding the four measured strands are presented.

The charts and HTML pages generated are very similar to those used in the previous study (Chapter 4). In the case of charts, the energy consumption of the CPU and RAM are represented by two vertical columns (the sum of both of them being equivalent to the total energy consumption measured). In turn, the execution time is represented by a line usually positioned near the top of the columns. In the case of HTML pages it is possible to consult the data obtained for each element in two different ways: graphically (chart) and numerically (table). In both cases the base information is the same, it only differs how the data are exposed.

In the rankings, beside to the four aspects measured, the various elements are also classified in relation to the ratio between energy consumed and execution time. Each element has in addition to its overall classification for a particular measure, the sum of the various classifications that it obtained individually for the various cases. For example, if a tool is the best for the 12 programs for a given strand, then in this instance it will have the rank

1 and also the respective sum (12). Naturally this value proves to be redundant when the elements are analyzed for a specific program. However, it proves to be very useful for more global cases because it allows, for example, to verify the real difference between elements with similar positions. For each classification were created four versions that differ in the number of decimal places considered (0, 1, 2 or 3). The different versions allow to analyze with a greater or lesser rigor certain aspect, proving to be especially useful in the analysis of large sets with several benchmarks.

Throughout the analysis will be added more information that we consider pertinent about how certain values were obtained or what they represent in particular. In total, more than 1000 charts, tables and HTML pages were elaborated and also more than 100 rankings referring to information gathered. Not all are disclose in this subsection, but only a small part of that set for the purposes of exemplification and assistance of relevant aspects in the intended analysis. However, all the processed information is available in the repository of this study[88] and in the project website[89]. There the data are organized in several directories according to the processing performed and with the elements analyzed. It is also possible to consult some additional examples in Chapter A.

5.6.1 *Programs*

Resorting to project CLBG, 12 programs were selected for this study. These gather a set of interesting aspects for analysis since they fulfill all the necessary prerequisites. Within the several characteristics they include, the programs stand out for being highly optimized solutions based upon simple, but very challenging, problems and also because they are associated to different background areas and program fields. After being used with benchmarks in the presented experimental methodology, it is now possible to observe how the characteristics of each program behave when analyzed from the different intended points of view. The herein presented data cannot be directly compared to the data divulged by CLBG since distinct measures, inputs, hardware, OS, among others, are being used. However, with due reservations, these results may be interpreted as complementary information.

The analysis is carried out simultaneously using the processed data from a visual and numerical perspective (especially through the charts represented in the Figures 38-49 and the directory 1.1 from the repository). Conclusions regarding the results for each program are better perceived through the numerical analysis perspective, while tendencies and behaviors observed along the various profiles are more explicit using a graphical perspective.

88 https://github.com/david-branco/programmingtoolsenergyconsumption
89 www.di.uminho.pt/~gepl/OCGREC

In any of these cases, the information refers to the execution of all the compilation profiles for each program using the methodology presented within the intended strands. Part of this information is also summarized in Table 11, namely the minimum, maximum, percentage difference and average values obtained for each program.

Program	Time (s)	Energy (J)	CPU (J)	Memory (J)	Energy/Time (J/s)
binary-trees	3.355-7.214 53.5 % (5.186)	40.843-78.631 48.1 % (58.844)	37.606-72.630 48.2 % (54.286)	3.231-6.029 46.4 % (4.558)	10.860-12.264 11.4 % (11.576)
chameneos-redux	7.706-9.112 15.4 % (8.358)	154.74-184.101 15.9 % (166.098)	149.475-178.231 16.1 % (160.478)	5.172-6.156 16.0 % (5.620)	18.724-21.397 12.5 % (19.888)
fannkuch-redux	21.918-58.501 62.5 % (40.537)	232.244-658.129 64.7 % (443.155)	217.685-619.299 64.8 % (416.238)	14.553-38.874 62.6 % (26.917)	10.273-11.293 9.0 % (10.768)
fasta	6.101-10.848 43.8 % (8.439)	17.86-66.781 73.3 % (41.991)	13.612-59.476 77.1 % (36.190)	4.248-7.305 41.8 % (5.801)	2.892-6.184 53.2 % (4.622)
k-nucleotide	7.113-22.755 68.7 % (14.434)	81.043-245.822 67.0 % (159.531)	75.857-230.399 67.1 % (149.443)	5.186-15.641 66.8 % (10.087)	10.744-11.730 8.4 % (11.241)
mandelbrot	4.624-26.755 82.7 % (15.639)	42.022-320.07 86.9 % (184.416)	38.910-302.269 87.1 % (173.991)	3.111-17.806 82.5 % (10.425)	9.086-11.967 24.1 % (11.525)
meteor	0.046-0.089 48.3 % (0.068)	0.425-0.934 54.5 % (0.683)	0.393-0.874 55.0 % (0.637)	0.032-0.060 46.7 % (0.046)	9.239-10.678 13.5 % (9.967)
n-body	3.473-25.390 86.3 % (14.308)	40.966-307.182 86.7 % (172.280)	38.658-290.323 86.7 % (162.779)	2.307-16.859 86.3 % (9.501)	11.781-12.455 5.4 % (12.103)
regex-redux	13.781-14.485 4.9 % (13.872)	138.577-147.229 5.9 % (140.309)	127.337-135.470 6.0 % (129.019)	11.223-11.759 4.6 % (11.290)	10.041-10.201 1.6 % (10.114)
reverse-complement	9.941-12.537 20.7 % (11.826)	15.994-24.825 35.6 % (20.479)	8.366-15.736 46.8 % (11.812)	7.628-9.180 16.9 % (8.667)	1.423-2.224 36.0 % (1.729)
spectral-norm	2.438-7.156 65.9 % (4.734)	21.504-86.862 75.2 % (52.589)	19.884-82.113 75.8 % (49.444)	1.620-4.752 65.9 % (3.144)	8.817-12.149 27.4 % (10.392)
thread-ring	9.307-9.786 4.9 % (9.555)	89.245-93.161 4.2 % (91.362)	81.511-84.919 4.0 % (83.328)	7.525-8.520 11.7 % (8.034)	9.361-9.677 3.3 % (9.562)

Table 11.: Measurement results for all programs.

Starting the analysis by examining Table 11, the high diversity of results between the programs can easily be observed as well as many of the factors that differentiate them.

Considering the execution time, it turns out that *meteor* and *fannkuch-redux* (see Figure 25) stand out clearly from the other programs because they have respectively the smaller and higher obtained values. As for the remaining cases, it is observed that they have values at relatively close intervals, namely: minimum values between 2.4-13.8 seconds, maximums in the intervals of 7.1-14.5 or 22.8-26.8 seconds and in average their execution time range between 4.7 and 15.6 seconds. The maximum values pertaining to execution time are within the expected because, as previously mentioned, the inputs given to the programs were adjusted (when possible) in order to provide close values for their worst case while still being considered relevant to this analysis.

The improvements that each program present for their best cases come from their own characteristics, and the capabilities of the GCC optimization suites to interact with them. This factor is linked to the difference in percentage value between the minimums and maximums for each program, where *thread-ring* and *regex-redux* clearly stand out negatively and *mandelbrot* (see Figure 24) and *n-body* do so positively (82.7% and 86.3%). Regarding the remaining, their values place between intervals of 15.4%-20.7% (*chameneos-redux* and *reverse-complement*), 43.8%-53.5% (*fasta*, *meteor*, *binary-trees*) and 62.5%-68.7% (*fannkuch-redux*, *spectral-norm* and *k-nucleotide*).

By analyzing the joint energy consumption of the two considered components it was noted that, also for this subject, the *meteor* and *fannkuch-redux* stand out as the instances showing respectively the lowest and highest values. Not considering those two programs, the remaining minimum values interval is situated between 16.0-154.7J and the maximum values interval between 24.8-230.0J. The average consumption locates between 20.5-184.4J. Although no previous considerations were made for the programs inputs regarding their final energy consumption, there was an overall significant improvement for worst and best results, averaging 111.5J considering all programs and 91.2J upon removing the two most extreme cases. This context also illustrates high diversity on how much the programs worst and best results percentually vary, placing mostly within four distinct intervals: from 4%-6% (*regex-redux* and *thread-ring*), 16% (*chameneos-redux*), between 36%-54% (*binary-trees*, *meteor* and *reverse-complement*) and from 64%-88% (*fannkuch-redux*, *fasta*, *k-nucleotide*, *mandelbrot*, *n-body* and *spectral-norm*). As well as observed for execution time, the four cases that stand out in this matter are again the pairs *thread-ring* - *regex-redux* (4.2% and 5.9%) and *mandelbrot* - *n-body* (86.7% and 86.9%).

Unsurprisingly, *meteor* and *fannkuch-redux* are also noticeable due to the same reasons for the individual consumption of the processor. Still, the results for the remaining programs cover a broad spectrum of values, specifically between 15.7-302.3J, 8.4-149.5J and 11.8-174J respective to maximum, minimum and mean values. A large discrepancy is also observed

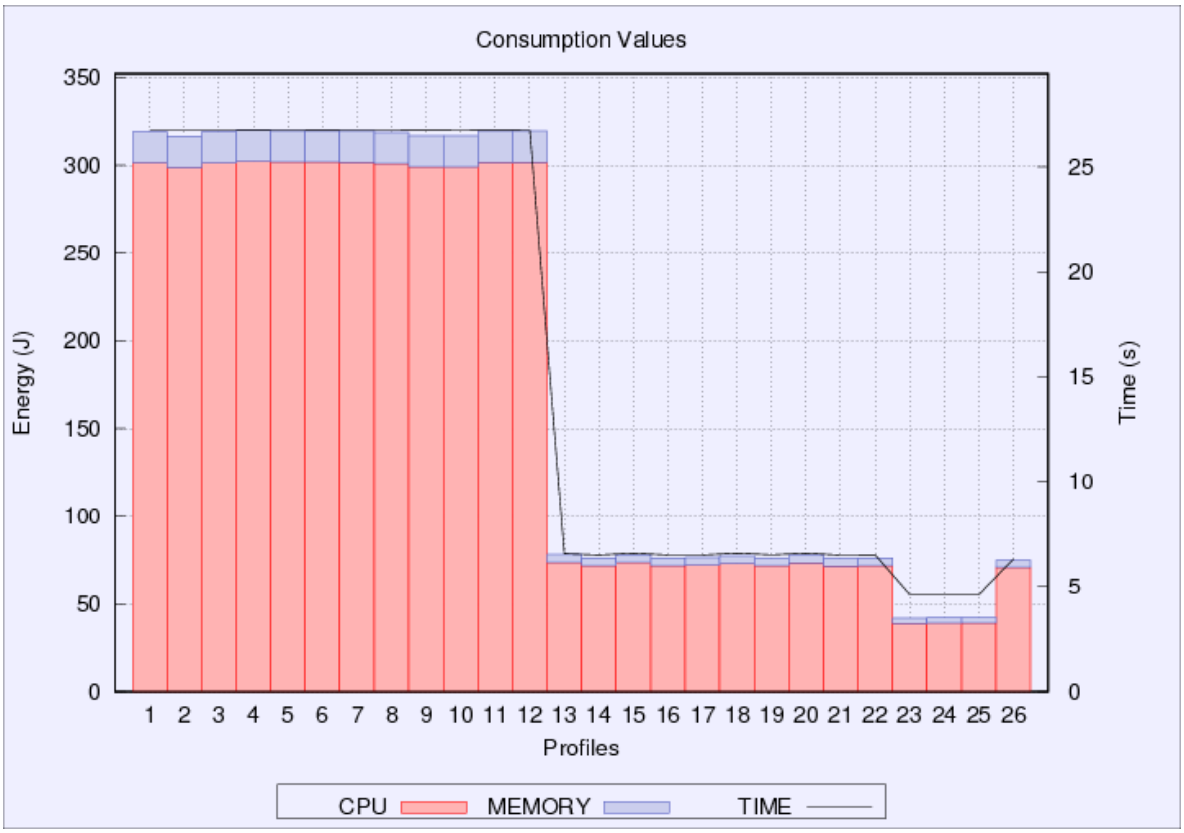


Figure 24.: Results of mandelbrot measurements.

between the extreme absolute values for each program, namely an average of 105.2J considering all programs or 86.0J after excluding the two most extreme results.

Comparing the energy consumption for this component with the previously considered total, it can be stated that it constitutes about 90% of the total consumption. Upon choosing a value which is closer in most programs (which means excluding the highest and lowest result), it is observed an even more substantial value, nearing 93% of the total. However, for the particular case of *reverse-complement* such tendency was not obtained since the relative consumption for this component was found near 58% (32% below average).

The proximity tendency between processor and total energy consumption is maintained in terms of the percentual difference between maximum and minimum values obtained for each program. By grouping the programs through intervals of proximate values, the resulting subsets prevail the same as before and inclusively most show fairly similar values: 4-6% (*regex-redux* and *thread-ring*), 16% (*chameneos-redux*), 46-55% (*binary-trees*, *meteor* and *reverse-complement*) and between 64-88% (*fannkuch-redux*, *fasta*, *k-nucleotide*, *mandelbrot*, *n-body* and *spectral-norm*).

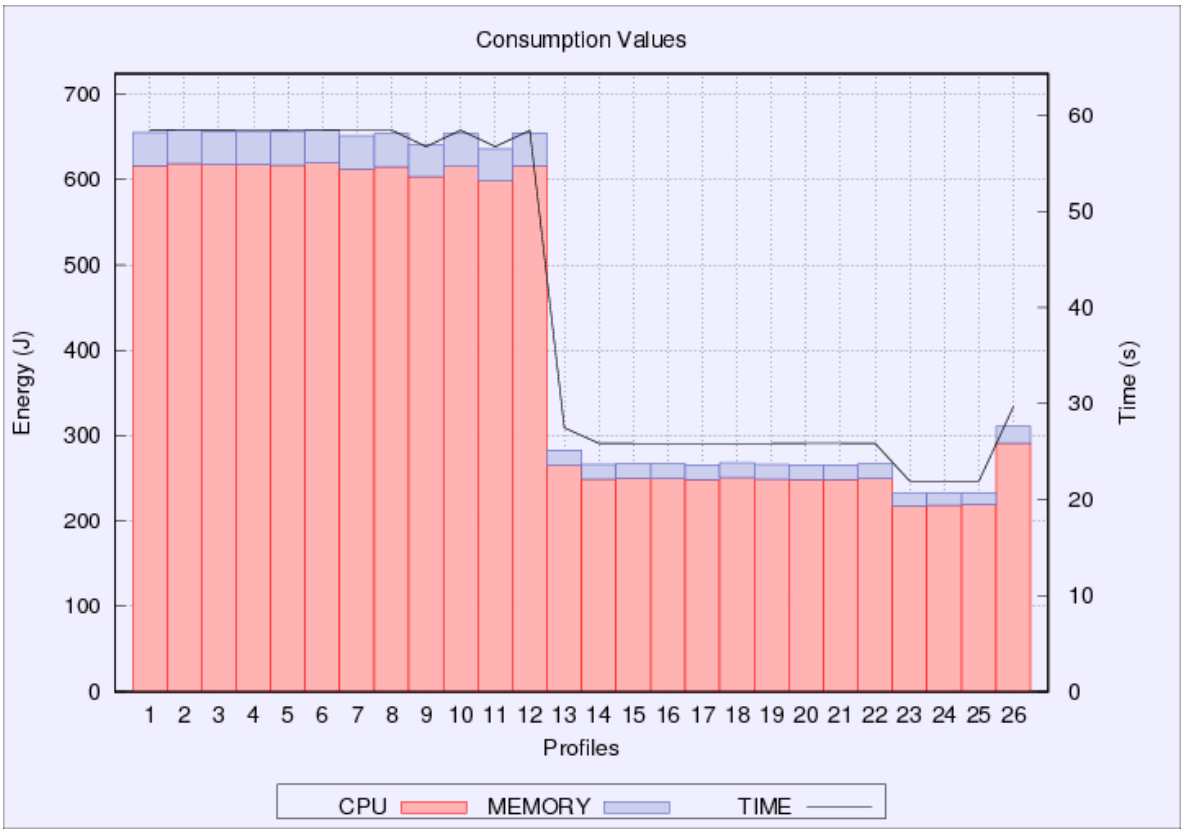


Figure 25.: Results of fannkuch-redux measurements.

Concerning the memory energy consumption, the results are complementary to the ones obtained for the processor, as expected. The same programs obtained the maximum and minimum results, and for the remaining programs are situated within the intervals 1.6-11.2J, 4.8-17.8J and 3.1-10.4J for minimum, maximum and mean values, respectively. In terms of energy consumption percentage for this component, *reverse-complement* (see Figure 26) is noted to, once again, be an outlier by gathering 42% of the total consumption, a value 35% above the average 7% found for the remaining programs ranging between 3-14%.

While the percentual differences between maximum and minimum results for the various programs present less proximity in terms of their intervals, they may still be equally clustered to four distinct subsets: 5% (*regex-redux*), between 11-17% (*chameneos-redux*, *reverse-complement*, *spectral-norm*), between 41-47% (*binary-trees*, *fasta* and *meteor*) and between 62-87% (*fannkuch-redux*, *k-nucleotide*, *mandelbrot*, *n-body* and *spectral-norm*). Although less prominent than observed regarding the processor, the percentual differences between memory and total energy consumption show some similarities in terms of values and behavior.

The energy consumption to execution time ratio overall suggests that the differences between best and worst cases for each program are not very significant in absolute terms. *Reverse-complement* and *chameneos-redux* stand out not only as providing the highest and lowest values, but also for being clear outliers, along with *fasta*, in several of the considered aspects. This information is distinctly perceptible upon analyzing the clustered values for each program, namely: minimum values between 1.4-2.9J/s for *reverse-complement* and *fasta*, 18.7J/s for *chameneos-redux* and between 8.8-11.8J/s for the remaining programs; maximum values of 2.2J/s for *reverse-complement*, 6.2J/s for *fasta*, 21.4J/s for *chameneos-redux* and the remaining between 9.7-12.5J/s; mean values between 1.7-4.6J/s for *reverse-complement* and *fasta*, 21.4J/s for *chameneos-redux* and between 9.7-12.5J/s for the remaining programs.

In terms of the percentual difference it is noted that some value fluctuation occurs among the various programs, yet mainly within a low and proximate range. *Regex-redux*, *thread-ring* and *n-body* display values below 5.5%, *k-nucleotide*, *fannkuch-redux*, *binary-trees*, *chameneos-redux* and *meteor* between 8.4-13.5%, *mandelbrot*, *spectral-norm* and *reverse-complement* between 24-36% and *fasta* 53.2%. This data allows the conclusion that energy consumption to execution time ratio shows the least percentual difference between the extreme cases for each program. Overall, the programs reached a maximum difference of 13.5% and averaged 8.1% (two programs even improved below 3.4%). Yet, four programs improved over 24% between worst and best result, and *fasta* actually doubled that value.

The analyzed data show there generally exists some diversity in program behavior along the several approached profiles and subjects. Globally, some difference can be found between the lowest and highest results for the several variables, indicating some oscillating behavior of the programs along the various profiles. The proportion of possible improvements is settled by the percentual difference between such cases and varies significantly between programs.

Through analysis of the graphical representation for each program it is also possible to confirm the mentioned tendencies. Some diversity indeed exists in the displayed behavior of the programs along the various profiles and subjects.

Thread-ring and *regex-redux* (see Figure 27) differentiate from the rest since they show very similar results regardless of the chosen profile. Such behavior precisely reflects the small percentual difference observed for execution time and energy consumption in both cases. The remaining programs exhibit a far more dynamic behavior throughout their profiles, remarking the existence of different types of fluctuations for the represented lines and columns instead of a single practically continuous behavior. In particular, the significant reduction between worst and best cases in terms of execution time and energy consumption for most programs is notorious, and visually it is verified that there are, indeed, consider-

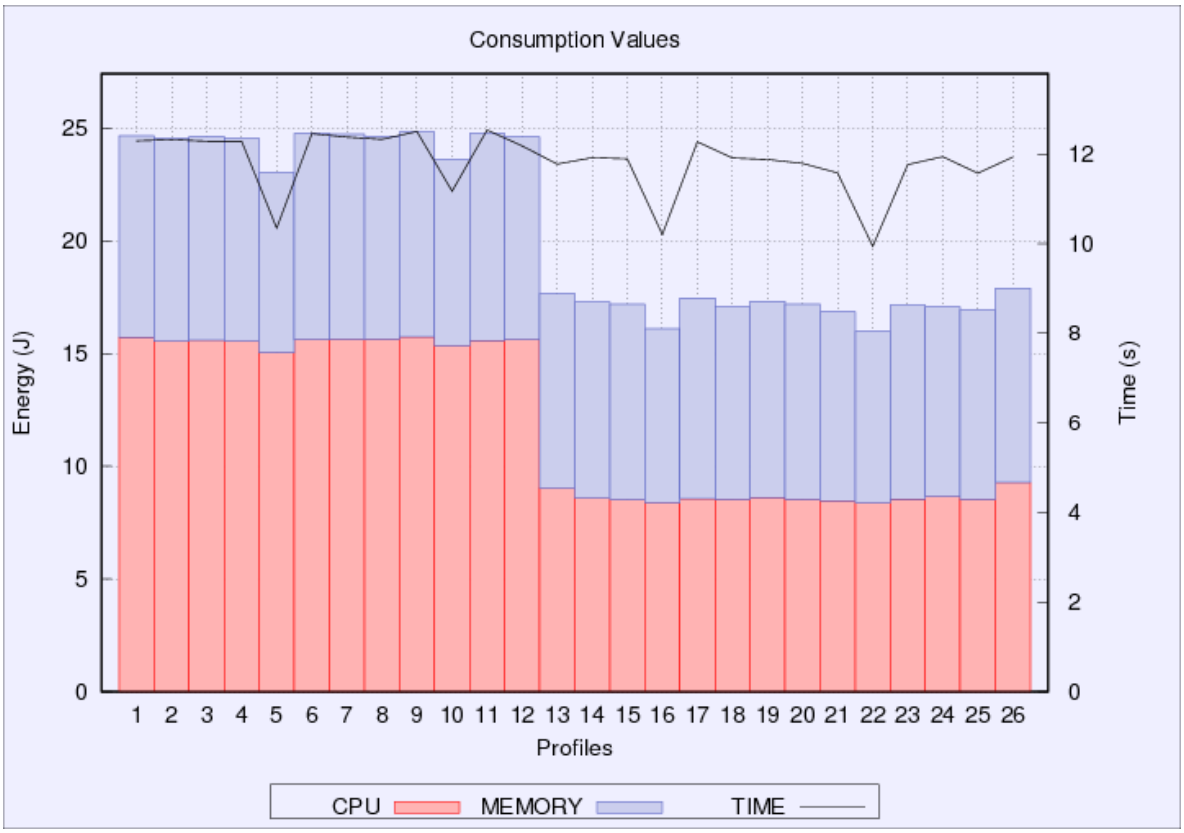


Figure 26.: Results of reverse-complement measurements.

able differences between the several profiles. Although a homogeneous behavior between these 10 programs cannot be observed, there are similar patterns and characteristics between them, which in practice reflect the previously considered clusters.

Another conclusion made clear through graphical representation analysis is the high energy consumption of the CPU in relation to memory. In fact, it is plainly demonstrated by the different proportions of the columns that the CPU energy consumption is responsible for over 90% of the considered global consumption. In addition, it is observed that the memory energy consumption generally presents very reduced values, even for the worst cases. *Reverse-complement* is the only exception regarding these considerations since the columns display average differences of only 15.4%, and inclusively the presence of profiles in which the memory exceeds the CPU regarding energy consumption.

Conjointly examining some of the considered strands, not only allows to observe how the programs generally behave, but also which characteristics and tendencies occur simultaneously. A relevant indicator in such matter constitutes in identifying in which way evolve the percentual differences between maximum and minimum results for the distinct elements.

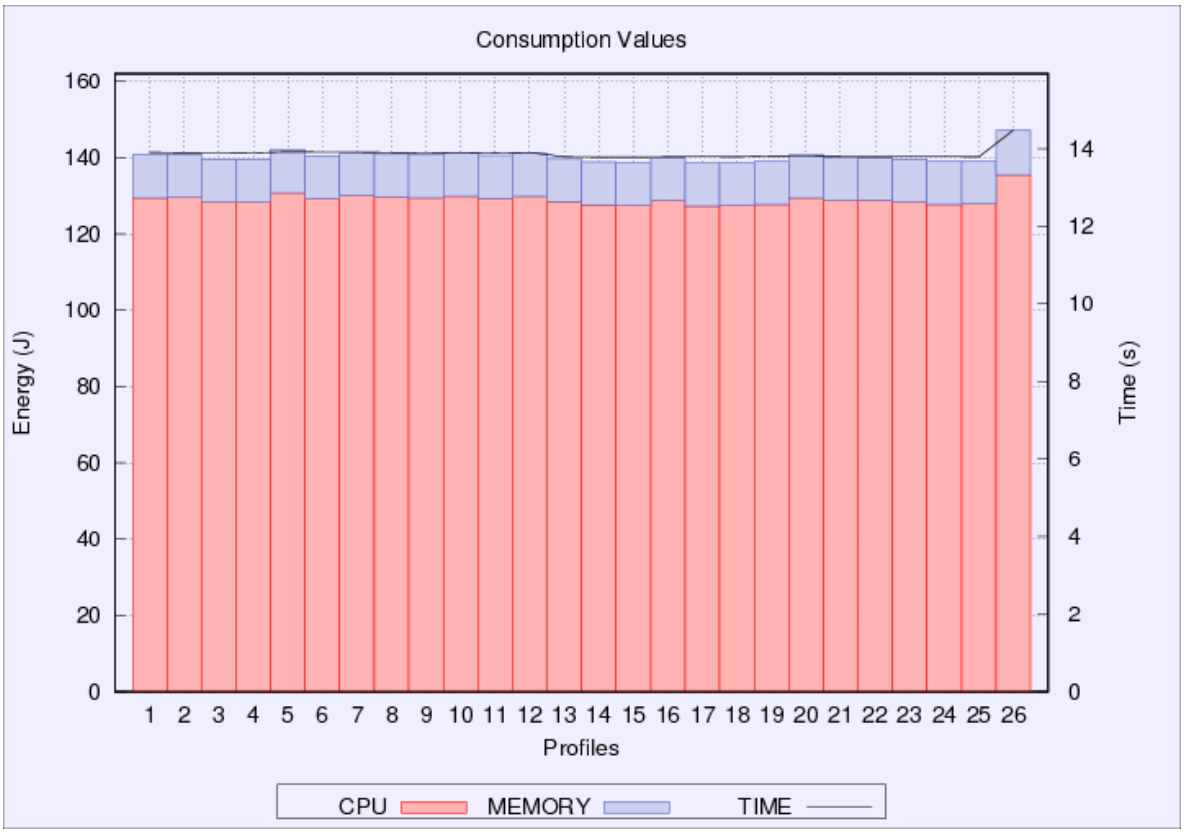


Figure 27.: Results of regex-redux measurements.

An overall correlation of tendencies is observed between execution time and total energy consumption along the various profiles. Such aspect is demonstrated through the proximity of percentual gains between the aforementioned measures (6.3% in average) and also due to half the programs displaying improvements lower than 2% for both measures, and only *spectral-norm* (9.3%), *reverse-complement* (14.9%) and *fasta* (29.5%) show values higher than 7%. Graphically the same conclusion may be achieved in virtue of the practically constant accompaniment between the execution time tendency line and energy consumption columns along the represented profiles. Although in some cases this behavior is not observed, such as in some *reverse-complement* profiles, the amount of those instances is minor considering the total amount of studied cases.

Stepping to further detail, the percentual difference between the two measures show that highest gains, although reduced, were obtained in terms of energy consumption than in terms of execution time. This occurs for 9 out of the 12 considered programs, and *binary-trees*, *k-nucleotide* and *thread-ring* were the only cases in which the opposite behavior was observed (with very low values). For all considered programs there was an average

improvement of 46.5% of execution time and 51.5% of energy consumption, resulting in a 5% difference between both measures.

After globally analyzing the data concerning execution time and energy consumption of the studied programs, it is concluded: there are significant improvements between best and worst results clearly demonstrating that profiles and compilation parameters are very influential in both subjects; there exists an evident equivalence between tendencies and values present throughout the various profiles demonstrating a narrow connection between both elements as well as the options that lead to the presented behavior changes; overall, higher percentual improvements (although possibly residual) of energy consumption over execution time were obtained per program, for the same compilation profiles.

Analyzing the energy consumption of the processor and memory (globally and locally), some interesting behaviors may be distinguished.

Such as previously observed, for most programs the CPU is responsible for the highest portion of the total energy consumption. In fact, comparing both total and processor energy consumption a significant proximity between maximum and minimum obtained values is found for each program, as well as for the percentual improvement between both cases. Also, there's an average difference of 1.5% between total and processor percentual energy consumption values, and for 8 of the programs the difference is found below 0.3%. Another observation is that, even though subtly, the difference favours the processor energy consumption for 11 of the considered programs. This information allows the conclusion that, overall, slightly more significant improvements are obtained for the processor comparatively to global energy consumption.

The close proximity of the presented behavior also allows to extend some of the conclusions stated in the previous paragraph. One such example, even though less noticeable, is the presence of a correlation between processor energy consumption and execution time. These behaviors are also clearly illustrated in the presented charts.

While generally the memory energy consumption is significantly reduced in relation to global energy consumption, a correlation of the displayed behaviors may also be found between these measures. In fact, for most cases the tendencies of both measures run proportionally. However, it may also be confirmed that this relation is not as narrow and dependent as the one verified for the processor and, inclusively, there exists a higher amount of exceptions and fluctuations within the obtained values.

These facts are demonstrated, for example, through the analysis of the percentual difference between extreme cases for the three distinct energy consumption strands that were considered. The average improvement obtained for the memory was 45.7%, 52.9% for the processor and 51.5% for the global energy consumption of all programs. Hence, it may be

stated that the three strands are proximate, but less accentuated for the memory energy consumption.

This data also highlights the similarity between the execution time (46.7%) in relation to the memory energy consumption. This observation meets the proximity previously observed between these two strands, namely between the subsets of programs that display values in proximate intervals.

Particularly evidencing the values of the four programs in which the differences show the highest fluctuation between these measures, an average 44.7%, 63.7%, 42.8% and 59.7% are obtained for execution time, processor, memory and total energy consumption, respectively. The same behavior can be observed in the chart that represents the results for the program with the highest memory energy consumption (*reverse-complement*, Figure 26).

All results and indicators permit the conclusion that, usually, the program execution time is directly related to its energy consumption (and vice-versa), and specially with the memory energy consumption. This reasoning is indeed coherent, and matches the general knowledge that memory is a relevant factor that influences the execution time of a program. Given the large speed difference between processors and memory, it may be assessed that memory related operations are precisely the performance critical point for some workloads. In this sense, and even though it may appear not to be a significant factor considering the whole system, memory energy consumption can influence the execution time of the program and consequently influence the global energy consumption. These results also suggest that improvements are generally more difficult to obtain for memory energy consumption than for processor and global energy consumption.

The results for the ratio between energy consumption and execution time mainly illustrate that the differences, either for each program or between programs, are not very significant. The mean difference between the extreme cases for the various programs is 1.6J/s, and they mostly show values fairly close to the mean case (10.3J). Another conclusion is that, substantially, significant improvements between best and worst results are not met. The detailed analysis of this variable proves relevant in situations in which a trade-off between energy consumption and execution time is intended. The selection of parameters that allow to reduce/increase this variable provides the developer with the ability to produce more balanced and efficient code for specific program cases. Namely, for situations in which one of the ratio elements is not a very limiting factor in the system, or yet for instances that require optimized energy consumption while maintaining an acceptable execution time (and vice-versa).

The discussed results reflect also the type of programs that were chosen for this study. Besides the proficiency of the compiler optimization, the large results variations observed for the different cases express the vast complexity and efficiency of the implemented solutions. Although energy consumption is not considered for the CLBG project, the several observed correlations suggest that the implemented optimization extend to this factor.

The different percentages of energy consumption between processor and memory also contemplate some of the characteristics of the presented problems and solutions. The formulation, purpose and background of each problem influence, for instance, which type of algorithms or hardware operations are applied. In case the solution requires costly operations such as writing large amounts of data to memory, or CPU intensive algorithms, naturally the generated results will be considerably influenced by these approaches.

5.6.2 *Programs - Tools*

From the performed research, it was obtained a result of 18 tools (15 Integrated Development Environments and 3 Build Automation Tools) that were considered suitable for this study. As it was already mentioned before, some of these tools provide options that may appear partially or completely in others as well. Even though the final calculated values are not distinct for these particular cases, they will as well be presented for all the analyzed tools.

In order to be comparable, for each tool an arithmetic average was calculated between the obtained values for each program and the total number of profiles that they provide. This information is used afterwards to elaborate rankings (Tables 16-19) (repository directories 2.3 and 2.4) and charts (Figures 50-61) (repository directories 2.2 and 2.3) that are crucial to deepen the research for this subsection. The analyzed aspects of each program are the same as previously stated. Exceptionally, for all rankings, multiple versions were created depending on the decimal places contemplated. This made possible to analyze the tools using different perspectives and, according to the objectives, make the evaluation more or less strict.

Even though the generated charts intend to contemplate a different analysis perspective, it is anyway concluded that the same previously noted characteristics are observed for each program. This occurs due to the considered data being small sets of individually obtained values.

Generally, it is verified that *thread-ring* (Figure 28) and *regex-redux* have a similar behavior and it is practically constant in all situations. In contrast, for the other programs it is observable some fluctuation in multiple considered situations and tools and, as a gen-

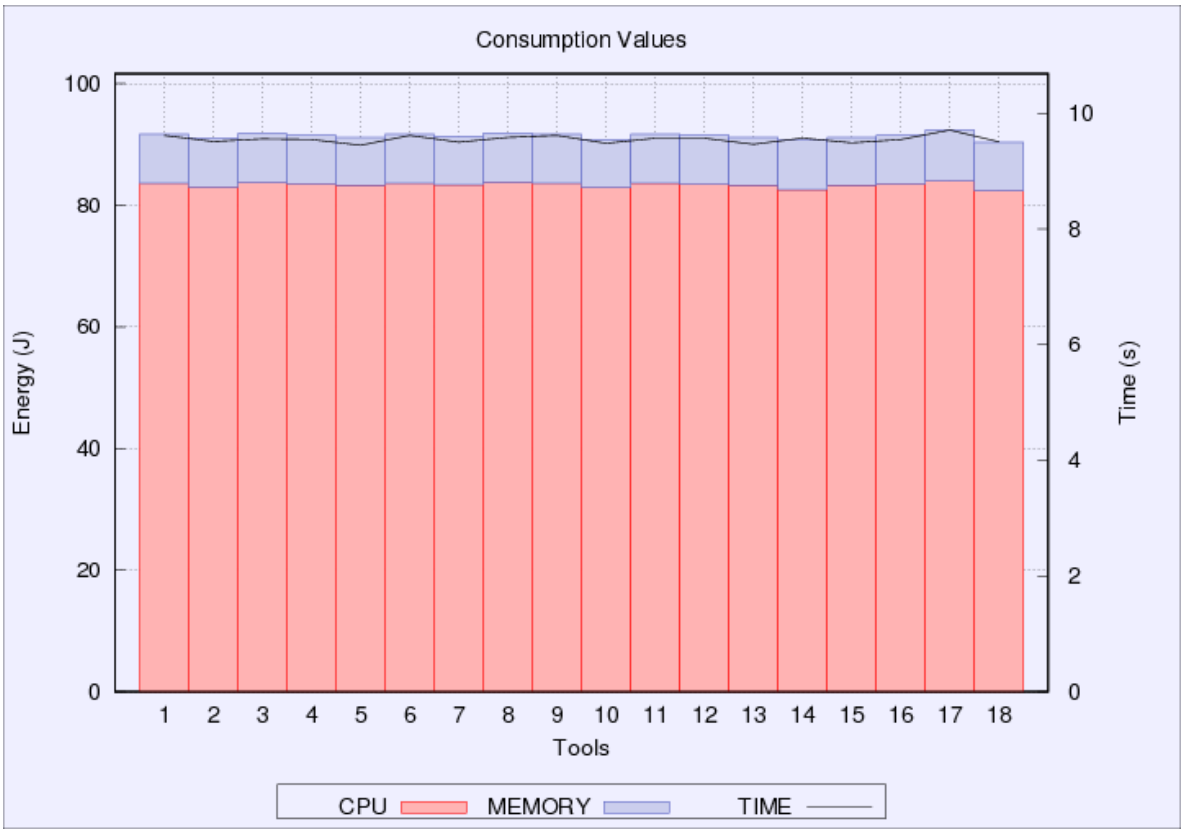


Figure 28.: Tools measurements for thread-ring.

eral rule, a similar behavior but with different proportions. In the *chameneos-redux* case, these variations are much less significant comparing with the remaining cases. Programs like *mandelbrot* and *n-body* (Figure 29) are exactly the opposite as they are responsible for the major differences between the best and worst-case scenarios. *Fannkuch-redux* and *k-nucleotide* charts show the best proportional growth considering all the aspects examined for the multiple tools. In *fasta* case it is possible to see the biggest variation between the execution time and memory energy consumption improvements upon comparing with the other two analyzed fields. At last, *reverse-complement* (Figure 30) stands out as the program with the biggest memory energy consumption, and it is observable, as well, in a constant execution time line.

Analyzing the charts in a tool viewpoint, it is easily contemplated that there are very distinct results between them. The ranking position and order of all the tools can be observed in Tables 16-19. Excluding some small nuances and position changes, it is verified that the global ranking plainly expresses the tools behavior for each individual program. This is made even clearer upon the based perspective being less rigorous.

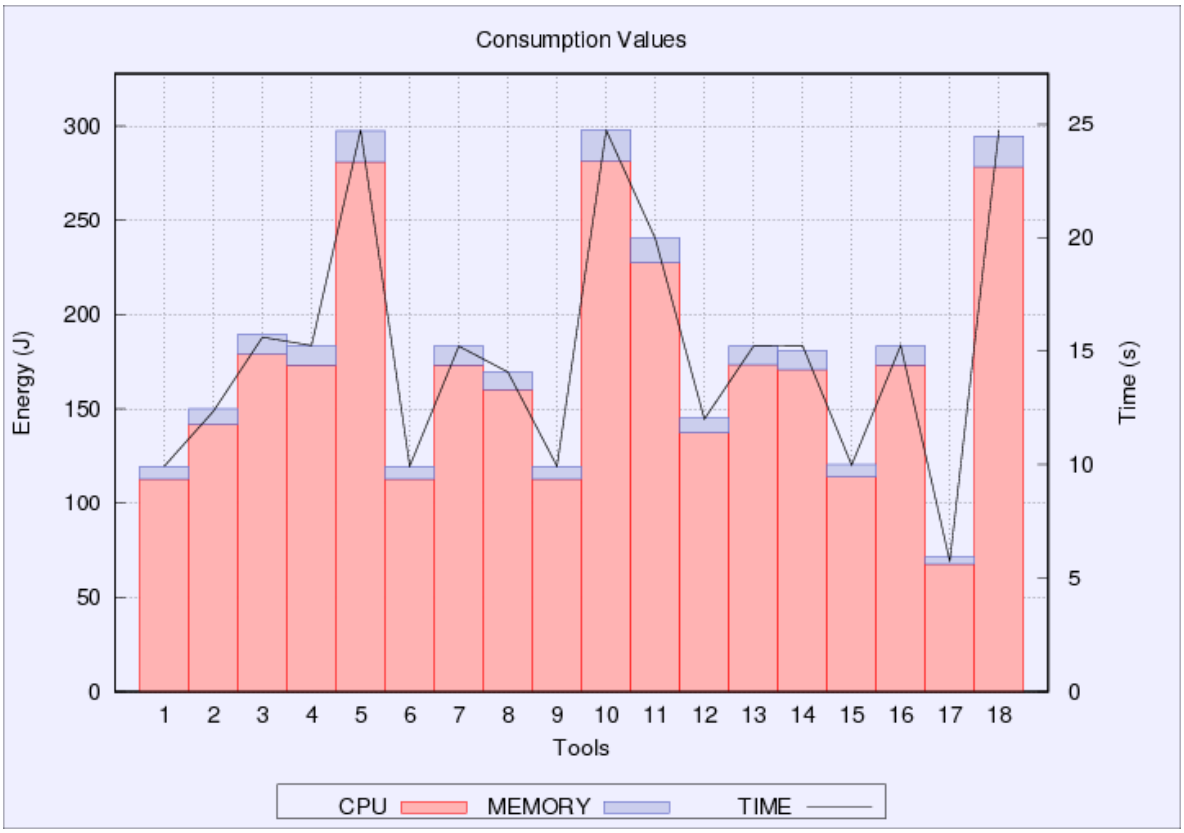


Figure 29.: Tools measurements for n-body.

It is worth remembering that some tools do not have self compilation profiles and opt to integrate other tools options. Besides the *Qt Creator* case, *CLion* and *KDevelop* apply the same options as *CMake* and the *Oracle Developer Studio* are heavily based on *NetBeans IDE*. For these examples, it is naturally concluded that the final results are exactly the same for both charts and rankings positions.

Based on a less strict point of view of the global ranking (e.g. Table 12), it is verified that, in the majority of the situations, the tools occupy similar ranking positions for the different calculated aspects and without any simple significant jumps. In this aspect the biggest exception is *ZinjaI* and for the remaining cases the most frequent changes occur on 0, 1 or 2 positions. This allows to understand that the tools provided options are indeed very generic, and that there does not exist a tool that is more suitable for a specific goal or program.

Examining the same ranking, it can be realized that *Sphere Engine* and *GPS* are the most efficient tools for the considered objective. On the other hand, *AWS Cloud9*, *Code::Blocks* and *Geany* stand out negatively. From both observations, curiously it turns out that the two

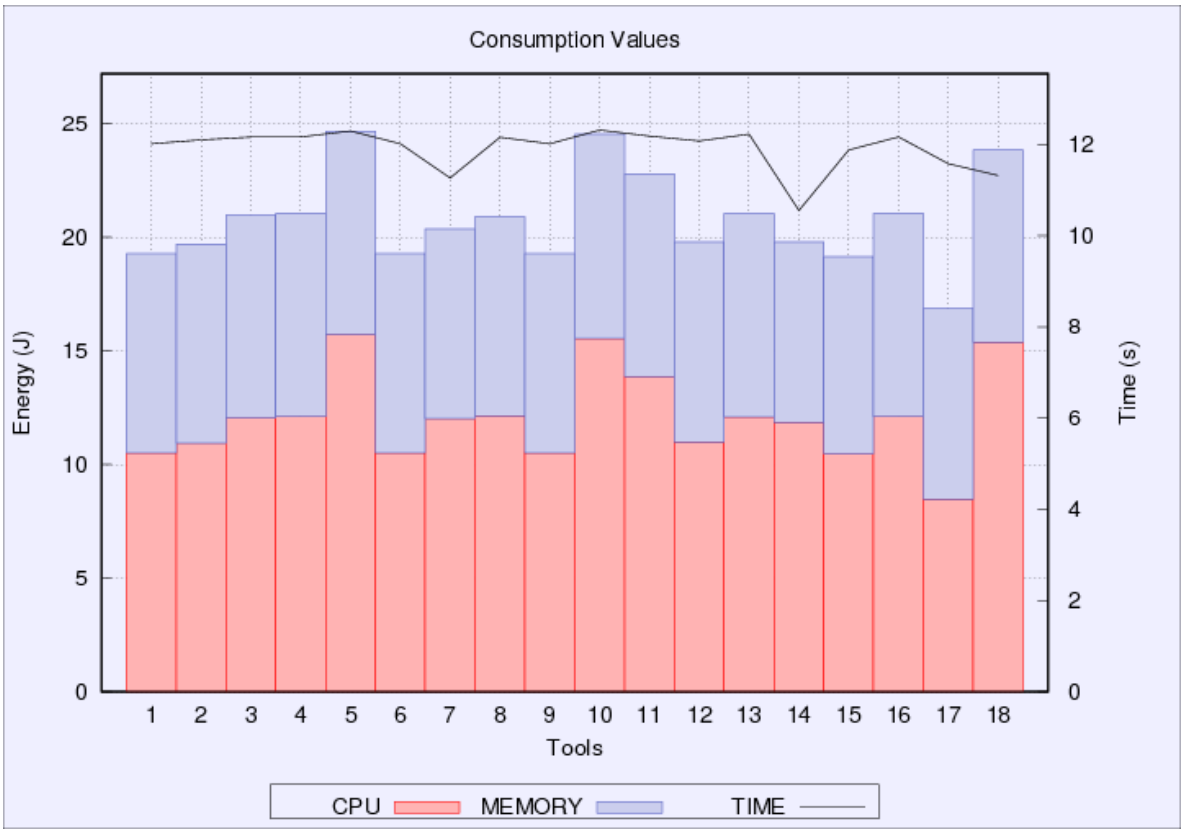


Figure 30.: Tools measurements for reverse-complement.

analyzed Cloud IDEs do not only show very distinct results, but are also displayed within the ranking top opposite positions.

Relative to BATs, and considering the global and individual rankings, it is verified that *CMake* and *qmake* are found, for most cases, in high and consecutive positions. General rule is that CMake provides better results than *qmake*, but there are exceptions in which the opposite happens (e.g. *spectral-norm*, *fasta*, *regex-redux* and *chameneos-redux*). The less efficient element is *Qbs* but the results are still within the average. BATs reveal themselves as good options within the studied subject. Qt Creator, because it integrates three described tools, is associated with an intermediate value between them. Although this is a good result, it can be better depending that the user opts only for a single integration, excluding *Qbs*. It should also be noted that the *Qt Creator* chart, for the already mentioned reasons, can be used to compare simultaneously all the studied BATs.

Considering the other IDEs and excluding the ones that only integrate external options, in general, it can be concluded that they all produce very similar results. *Eclipse CDT* and

Tool Name	Execution Time (s)	Total Energy (J)	CPU Energy (J)	Memory Energy (J)	Energy/Time (J/s)
CMake	3 (41.2)	3 (101.2)	3 (96.2)	3 (39.8)	3 (46.8)
qmake	8 (45.3)	4 (105.7)	4 (99.3)	6 (42.3)	4 (47.0)
Qbs	11 (50.5)	9 (124.5)	9 (119.5)	11 (49.5)	7 (50.0)
NetBeans IDE	10 (49.5)	11 (127.0)	9 (119.5)	10 (47.5)	7 (50.0)
Code::Blocks	15 (65.0)	15 (179.0)	14 (174.0)	15 (60.0)	11 (59.0)
CLion	3 (41.2)	3 (101.2)	3 (96.2)	3 (39.8)	3 (46.8)
CodeLite	6 (44.0)	7 (118.0)	7 (114.0)	7 (42.5)	8 (50.5)
Eclipse CDT	5 (43.5)	6 (112.5)	6 (108.0)	5 (42.0)	6 (49.0)
KDevelop	3 (41.2)	3 (101.2)	3 (96.2)	3 (39.8)	3 (46.8)
Geany	14 (63.0)	13 (170.0)	12 (165.0)	14 (59.0)	12 (62.0)
Anjuta DevStudio	12 (57.2)	12 (150.5)	11 (143.0)	12 (54.5)	9 (52.0)
Qt Creator	7 (44.7)	5 (107.9)	5 (102.4)	8 (42.8)	5 (47.6)
DialogBlocks	9 (46.0)	10 (125.5)	10 (120.0)	9 (44.5)	9 (52.0)
ZinjaI	4 (42.0)	8 (121.5)	8 (116.5)	4 (40.5)	10 (58.0)
GPS	2 (38.0)	2 (96.2)	2 (91.5)	2 (35.8)	2 (46.5)
Oracle Developer Studio	10 (49.5)	11 (127.0)	9 (119.5)	10 (47.5)	7 (50.0)
Sphere Engine	1 (33.0)	1 (81.0)	1 (79.0)	1 (34.0)	1 (44.0)
AWS Cloud9	13 (61.0)	14 (178.5)	13 (173.0)	13 (55.5)	13 (63.5)

Table 12.: Tools ranked with 1 decimal point.

Anjuta DevStudio can be highlighted for the positive and negative side, respectively. It is possible to conclude then that, besides situations in which a specific IDE is necessary, the choice can be made without any significant performance penalty.

Analyzing the multiple tables and the positions each tool occupies, some of the previous conclusions are again observed. There exists high proximity between positions for a given tool considering the variables of execution time - total energy consumption, total energy consumption - CPU energy consumption and between execution time - RAM energy consumption. The same observations can be made through the each tool chart.

The presented information allows to deduce that some tools reached very close results between them. Given the high similarity of their final results, they display individual positions only for being different programs. A relevant and less rigorous analysis was then useful, as it contributed to grouping small sets of options with similar results. Considering this, and with the help of the chart and ranking information, the following lists were obtained respectively for the execution time variable and the energy consumption variable:

Time Ranking:

1. *Sphere Engine*;
2. *GPS*;
3. *CMake*, *CLion*, *KDevelop*;

4. *qmake*, *Qt Creator*;
5. *Eclipse CDT*, *ZinjaI*, *CodeLite*;
6. *Qbs*, *DialogBlocks*, *NetBeans IDE*, *Oracle Developer Studio*;
7. *Anjuta DevStudio*;
8. *Code::Blocks*, *Geany*, *AWS Cloud9*.

Energy Ranking:

1. *Sphere Engine*;
2. *GPS*;
3. *CMake*, *CLion*, *KDevelop*, *qmake*;
4. *Qt Creator*;
5. *Eclipse CDT*;
6. *Qbs*, *DialogBlocks*, *NetBeans IDE*, *Oracle Developer Studio*, *ZinjaI*, *CodeLite*;
7. *Anjuta DevStudio*;
8. *Code::Blocks*, *Geany*, *AWS Cloud9*.

Based on this list it is clear to observe some of the already mentioned aspects. Some tools stand out on top of each list, and it is also verified that the middle positions are occupied by the tools with average, and very similar, results. The resemblances between the lists are very noticeable, and both cases display short variations of the positions.

Expanding the method previously applied, it is possible to integrate in a single ranking both variables of execution time and total energy consumption. Using the same elements as used in the lists above, the following classification is obtained:

Time and Energy Ranking:

1. *Sphere Engine*;
2. *GPS*;
3. *CMake*, *CLion*, *KDevelop*, *qmake*;
4. *Qt Creator*;
5. *Qbs*, *Eclipse CDT*, *ZinjaI*, *CodeLite*, *DialogBlocks*, *NetBeans IDE*, *Oracle Developer Studio*;
6. *Anjuta DevStudio*;
7. *Code::Blocks*, *Geany*, *AWS Cloud9*.

It should be noted that this analysis is restricted to the calculated elements of the last section methodology. There are some other discarded considerations regarding central tool aspects, such as its target audience, amount of features and sophistication, usage complexity, generated code size, among others.

Beyond voicing the capabilities of each tool, this information shows specifically which objective each tool aims for. Within the multiple particularities and characteristics included in them, they provide the user with different amounts and types of compilation profiles. As already mentioned, in most cases the tool has two profiles: one with a simplified behavior (usually with warnings and debug options) and a more sophisticated one (containing optimization suites). However, it can be noted that there are tools which intents may cause side effects that are not necessarily related to the resulting value of the tool. For instance, some tools contain a greater number of sophisticated profiles that produce more efficient results, and ultimately that turns out to be the differentiating factor on the analysis. This situation will be approached within the following subsections.

Grossly analyzing the intention of each studied tool profile, the related charts and rankings confirm the observed result. Namely, it is perceived that the tools with the most optimization load occupy the best positions, while on the other hand, the ones that discard optimization options rank within the worst places. Considering only the most highlighted example, it is precisely verified that there are optimizations in the only profile that *Sphere Engine* contains and in 3/4 of all *GPS* profiles. On the contrary, *AWS Cloud9*, *Code::Blocks* and *Geany* present more basic profiles and *Anjuta DevStudio* provides optimization on a single profile within a total of 4. Concerning BATs, *CMake* and *qmake* contain 2 out of 3 profiles with optimization while *Qbs* follows the usual case, in which only half of the profiles have optimization.

Given the objective of analyzing the tool values effectively, rankings were elaborated exclusively for tools that follow the same and most usual behavior. For this purpose, seven tools were selected containing only two compilation profiles with opposite behaviors: one more simplified and another more advanced regarding the impact on the generated code. The following results follow the same methodology as used before:

Time Ranking:

1. *Eclipse CDT*, *CodeLite*, *ZinjaI*;
2. *DialogBlocks*;
3. *NetBeans IDE*, *Oracle Developer Studio*;
4. *Qbs*.

Energy Ranking:

1. *Eclipse CDT*;
2. *CodeLite*, *Qbs*;
3. *DialogBlocks*;
4. *ZinjaI*;
5. *NetBeans IDE*, *Oracle Developer Studio*.

Time and Energy Ranking:

1. *Eclipse CDT*;
2. *CodeLite*;
3. *DialogBlocks*, *ZinjaI*;
4. *Qbs*;
5. *NetBeans IDE*, *Oracle Developer Studio*.

This data allows the conclusion that it is possible to obtain a classification for the multiple tools based on the similar capabilities they offer to the user. However, it is also observed that there is a small difference between the obtained values for each one. Even though the rankings were created using less strict criteria, in practice, the difference between consecutive ranking positions can be only about some units of Joule or seconds. Although it is plausible to consider the rankings, the selection of different options does not result in a high performance impact regarding user experience.

5.6.3 *Tools - Profiles*

In total, the tools considered for this study provide 51 compilation profiles to the user. Each option is characterized for having a very specific purpose which is defined by several aspects, namely its name and its parameters. The variety of profiles provided by each tool varies in regards of diversity and sophistication of options intended to be used during the software development process. Some profiles intend to improve the user experience during compilation through the use of warnings, others intend to turn the code more efficient in terms of its size or performance, or yet to enable the debugging and profiling features for the code.

Through this study it can be verified that energy consumption is not yet considered a primary factor for the tools while designing the compilation profiles. Yet, it is relevant to examine which is the impact that the analyzed profiles cause and in which way the tools are differentiated considering the intended profiles.

For each tool charts (repository directory 2.3) were generated to represent the results obtained for the compilation profiles of each of the analyzed programs. Further illustrations (repository directory 2.4) were also elaborated along with rankings (repository directory 1.4) referring to profile categories that are discussed throughout the present subsection. Based upon these elements, it is possible to draw more specific conclusions from the analysis of the measured data. The information presented in the previous subsections maintains valid, as well as complementary towards a global analysis.

The processed data reveals the overall existence of some diversity in the values obtained for each program, for each tool, and for each profile. However, as previously observed, the concrete quantification of such variation is highly dependent on context, objective and characteristics of the mentioned subjects.

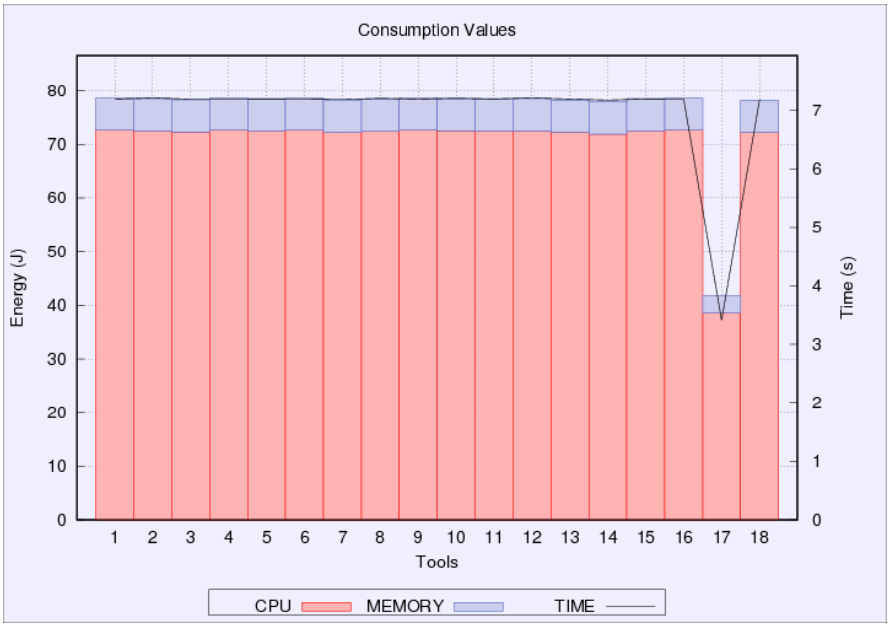


Figure 31.: Default profiles measurements for binary-trees.

Considering the presented elements, one of the first topics that may be approached is related to the default profile provided by the tools. This option does not correspond to the profiles named Default, but to the ones that are activated while the user does not alter any specification within the tool. As for the tools present in this study, they correspond to the profile indicated in the first position upon their description (Table 8). Since *Qt Creator* provides many options that depend on the various tools that may be used, the default profile of the most appropriate option (*qmake*) was selected. The results for the 18 corresponding compilation profiles are illustrated in the charts presented in repository directory 1.5 and also classified according to the rankings presented in directory 2.5 considering the various

intended subjects. Figure 31 shows a demonstrative example of the behavior of the majority of the charts obtained.

Tool Name	Execution Time (s)	Total Energy (J)	CPU Energy (J)	Memory Energy (J)	Energy/Time (J/s)
CMake	3 (22.2)	3 (42.5)	3 (43.5)	3 (20.2)	3 (19.0)
qmake	4 (23.0)	4 (43.0)	4 (44.0)	4 (20.7)	3 (19.0)
Qbs	9 (26.5)	7 (51.0)	7 (52.0)	10 (23.5)	5 (19.5)
NetBeans IDE	8 (26.0)	10 (54.0)	9 (55.0)	9 (23.0)	6 (20.0)
Code::Blocks	12 (34.0)	12 (79.0)	11 (81.0)	13 (32.0)	10 (23.0)
CLion	3 (22.2)	3 (42.5)	3 (43.5)	3 (20.2)	3 (19.0)
CodeLite	6 (24.5)	7 (51.0)	8 (53.5)	8 (22.5)	7 (20.5)
Eclipse CDT	6 (24.5)	6 (48.0)	6 (49.5)	5 (21.0)	5 (19.5)
KDevelop	3 (22.2)	3 (42.5)	3 (43.5)	3 (20.2)	3 (19.0)
Geany	13 (35.0)	13 (80.0)	13 (84.0)	12 (31.0)	10 (23.0)
Anjuta DevStudio	10 (30.5)	11 (65.0)	10 (67.5)	11 (27.5)	8 (20.8)
Qt Creator	5 (23.4)	5 (44.6)	5 (45.6)	6 (21.1)	4 (19.1)
DialogBlocks	7 (25.5)	8 (52.0)	9 (55.0)	9 (23.0)	5 (19.5)
ZinjaI	6 (24.5)	9 (53.0)	8 (53.5)	7 (22.0)	5 (19.5)
GPS	2 (21.5)	2 (40.2)	2 (41.0)	2 (18.8)	2 (18.8)
Oracle Developer Studio	8 (26.0)	10 (54.0)	9 (55.0)	9 (23.0)	6 (20.0)
Sphere Engine	1 (19.0)	1 (29.0)	1 (31.0)	1 (15.0)	1 (18.0)
AWS Cloud9	11 (33.0)	14 (80.5)	12 (82.5)	12 (31.0)	9 (22.5)

Table 13.: Default profiles ranked with 0 decimal points.

Taking into account the mentioned illustrations (e.g. Figure 31) and rankings (e.g. Table 13) it stands out that, in exception of one profile, all cases show very similar results regardless of the program. Considering the less rigorous global ranking in particular, it may be noted that in fact only small differences exist between those options. Such proximity is mostly due to the generality of the tools providing a very simple and generic profile as their first option. The intent of easing the initial stage of software development usually leads to being provided options that manifest low impact towards the behavior of the program (which may even be empty profiles) or that assist the compilation (e.g. warnings) and the compiled code analysis (e.g. debug and profile). The inclusion of optimization and other more sophisticated options is not a priority for tools at this stage, and inclusively, some applied changes may complicate or prevent some stages of the process (e.g. debugging).

The reason *Sphere Engine* acts as an exception in relation to the other profiles is precisely because an optimization suite exists in its default profile. This option allows the tool to obtain large energy consumption and execution time benefits for the programs. This approach is not used for any other studied case mainly because the tool only provides a single compilation profile that focuses on other aspects adequate to a product at a more advanced stage.

Hence it is concluded that for *Sphere Engine* it is the best option considering the defined scenario and priorities. If the user intends to select one of the remaining tools, there are no special considerations required since all present very similar results. For more extreme cases in which such requirement is needed, the presented ranking suggests using a BAT (or an IDE that integrates it) and discarding options such as *Code::Blocks* or *Geany*.

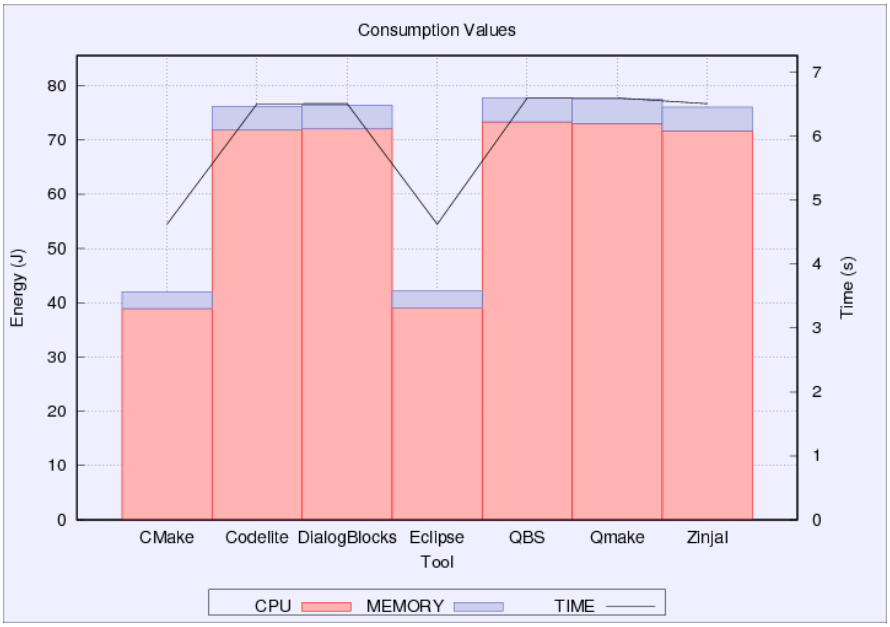


Figure 32.: Release profiles measurements for mandelbrot.

The profile names of the analyzed tools tend to suggest their characteristics and intended objectives. The 51 approached profiles exhibit a total of 10 distinct names, with high occurrence of the options Debug and Release (15 and 14 respectively). The charts contained in repository directory 2.4 present the results grouped according to the different definitions that each program possesses (Table 8). Using the same criteria, rankings in directory 2.2 were created for the two most recurring options.

There are 6 profiles that are found exclusively for a specific tool, hence, for those cases the charts only show a single option. From that information it is not possible to establish any connection, and as such, no consideration is made regarding the profiles *SomeOpt*, *RelwithDebInfo*, *Optimized*, *FullOpt*, *MinSizeRel* and *FullAutoInline*.

For the majority of programs some discrepancy is noted between the obtained values for the tools that provide the profiles Default and Profile. In both cases this is explained by the fact that the tools may or may not opt to include optimization in these elements. Since it

is a highly influential factor in the studied measures, naturally the two approaches lead to very different results.

For the case of the Release profiles (e.g. Figure 32) two types of behavior are observed. On one hand, for most programs the tools show practically constant and very similar values regarding the considered subjects. On the other hand, it is noted that the tools *Eclipse CDT* and *CMake* (and derived) are able to produce more efficient results than the remaining, for a considerable amount of programs. This occurs for 5 cases, and through graphical analysis it is verified that particularly *n-body* and *mandelbrot* show very significant differences (39.7% and 29.9% respectively). Both highlighted behaviors are also due to the use of optimization suites, specifically because of the different levels chosen by the tools. While it is indifferent for most observed cases, there are some situations in which tools that use the level 3 obtain better results than the ones that use other level.

Regarding Debug profiles (e.g. Figure 33), the tools generally display similar results considering the various analyzed subjects. Taking into account the program that, for this situation, displays the most fluctuating behavior (*chameneos-redux*), it is noted that the differences of values show low significance. This way it can be verified that, usually, the tools provide profiles that are equivalent in regards of the considered subjects upon providing Debug options to the user.

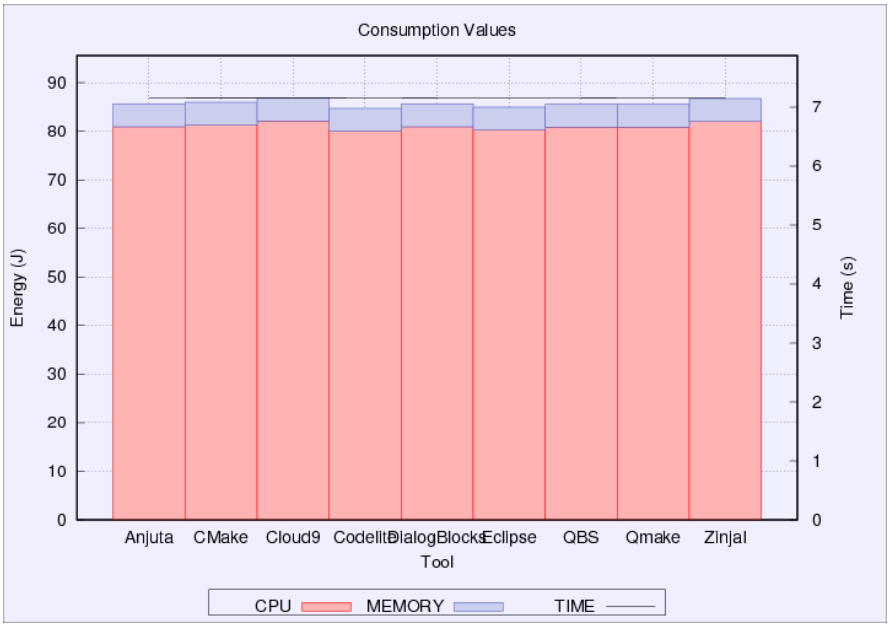


Figure 33.: Debug profiles measurements for spectral-norm.

5.6.4 *Profiles - Parameters*

On the Figures 38-49 (and repository directories 1.1 and 1.2) and Tables 20-23 (and repository directories 2.1 and 2.2) the obtained results for the 26 compilation profiles are presented. Additional material (repository directories 1.6) related to the same information but grouped by smaller categories, was also generated and will be analyzed along the subsection.

Some aspects related to the compilation profiles were previously examined either in tool or program viewpoint. In this subsection, the objective is mainly to analyze the results related to profiles and its parameter types: in what ways they differ and how much impact they really have on the considered aspects.

Regardless of different nomenclatures and without going into too much detail about the content of each profile, it is possible to conclude, based on the applied methodology, that there are very similar profiles in terms of calculated information. For example, using some of the charts in which the program behaves dynamically along the multiple profiles, it is easily verified that generally there are 3 to 6 levels of very close values. This clearly shows that it is possible to group profiles in categories with similar characteristics (in addition to the nomenclature). This approach allows both to understand in particular which impact they have on the calculated values for the subject, and to simplify the analysis through result redundancy reduction.

Following the analysis performed in the previous subsection, four distinct categories are easily highlighted that differentiate all profiles considering two of the most relevant aspects: (a) profiles without optimization and debug options, (b) profiles with debug but without optimization options, (c) profiles with debug and optimization options; and (d) profiles without debug but with optimization options. Using charts contained in repository directory 1.6 (e.g. Figures 34, 35, 36, 37) and presently disregarding the type and depth of the options contained in the corresponding profiles, three common properties immediately stand out:

1. The obtained values for the profiles of category (a) and (c) are generally very similar considering the studied aspects.

2. There is only a residual difference for the (b) category profiles. In particular, based on the presented charts, it is observed that the execution time line and energy consumption columns have an almost constant behavior for the first eight cases and only a single fluctuation on four result profiles.

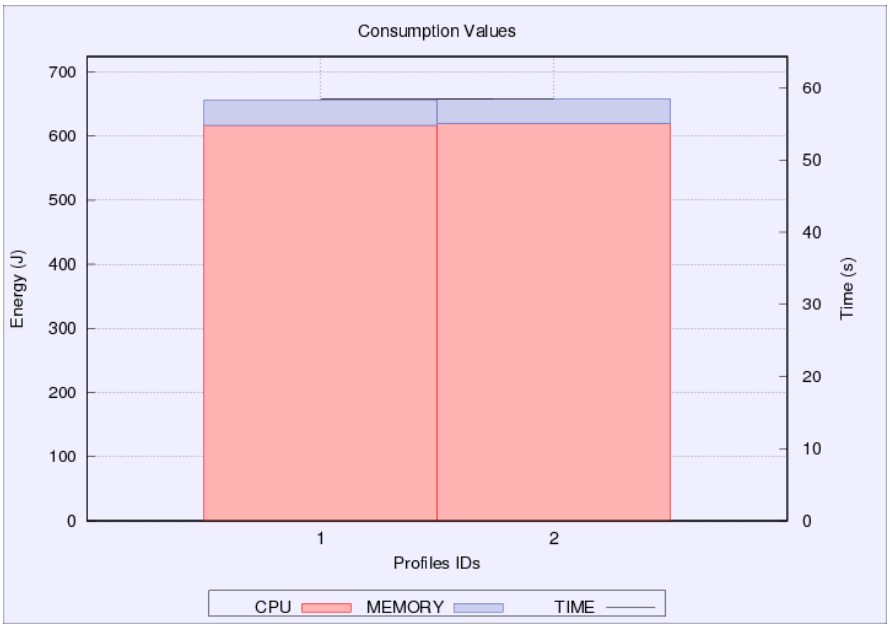


Figure 34.: Profiles without optimization and debug options for fannkuch-redux.

3. In general, a large variation on the obtained result between profiles with optimization can be observed. However, there exists no absolute definition on how it happens since each of those cases present some differences on their behavior and tendencies. Sometimes the columns indicate the format in an ascending escalator shape (*k-nucleotide*, *spectral-norm*) and in other programs the first and last profile present themselves as less efficient that the rest (*fasta*, *n-body*). Besides that, it can even be noted that the profiles *23* to *25* show lower (*fannkuch-redux*) or equivalent (*thread-ring*, *regex-redux*) results than the remaining cases.

In exception of cases in which the rigor is appropriate, the referred properties allow to conclude that the profiles of the (a) and (c) categories can be considered similar within the mentioned aspects. In these cases, it is indifferent to the user which is the best option if the only considerations are exclusively related to execution time or energy consumption of the programs.

Regarding the options contained on the (b) category, it is also observed that there are no significant differences. In fact, there are some profiles that allow to obtain slightly more efficient results for very specific cases. However, the improvements, for the majority of the target audience and programs, have no major impact or relevance. So, generally, it is possible to admit the profiles within this category are very similar.

For the (d) category options, it is not possible to elaborate such comprehensive conclusions. Given the large results variation, it would be necessary to further analyze which

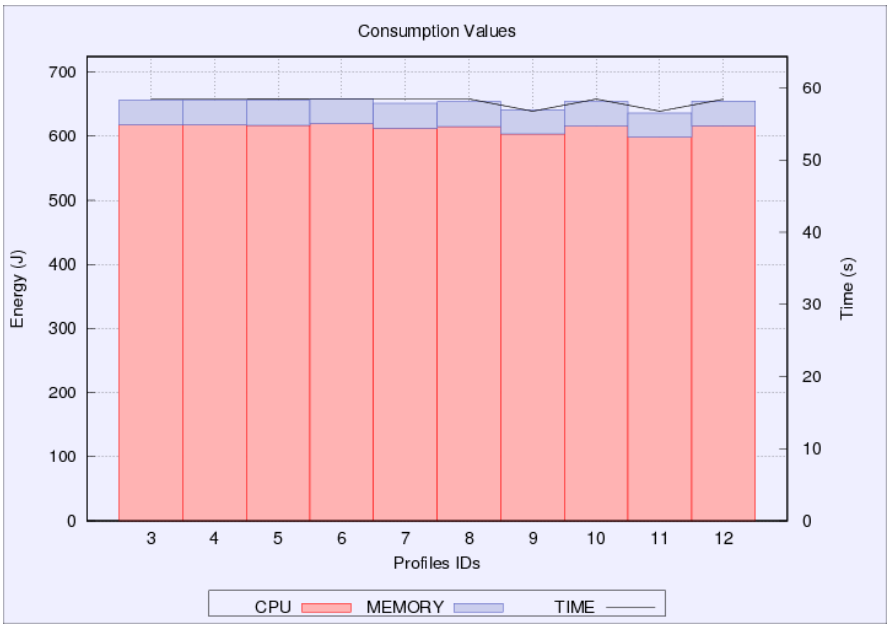


Figure 35.: Profiles with debug but without optimization options for fannkuch-redux.

parameters are used in each compilation profile. The obtained data renders impossible to find an example that is better or worst that the rest, considering all programs. So, bearing this consideration, it is plausible to conclude that the optimization application efficiency vastly depends on the program type and its respective source code.

During the study some comparisons were made between the category options based on the determined parameters. In detail, multiple instances of the clear difference between options with and without optimization (respectively category (d) with (a) and (b)) were observed. However, a more detailed analysis is required in order to understand how profiles with only debug (b) or optimization (d) differentiate from the hybrid ones that contain both characteristics (c)). These are called *RelwithDebInfo* and are known precisely for containing a moderate level of optimization along with management options of debug information.

Comparing the produced results for the (b) and (c) categories, it is obvious that there is a large difference between both situations. It is again visible that options with optimization are more effective, in all aspects, than the others that include other category options.

The options of (c) category show very similar values between themselves and some (d) category profile cases. It is generally observed that there exists large proximity in the produced results between the profiles with IDs from *14* to *15* and from *16* to *22*. Closely analyzing which common element types cause this behavior, it is possible to conclude that most of them use a *-O2* optimization suite, and the profiles from both categories that contain

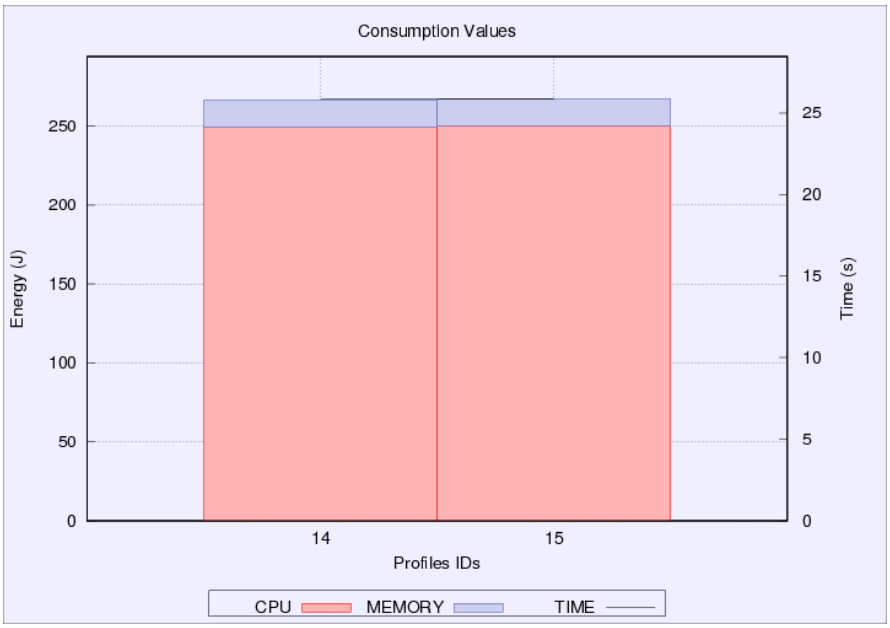


Figure 36.: Profiles with debug and optimization options for fannkuch-redux.

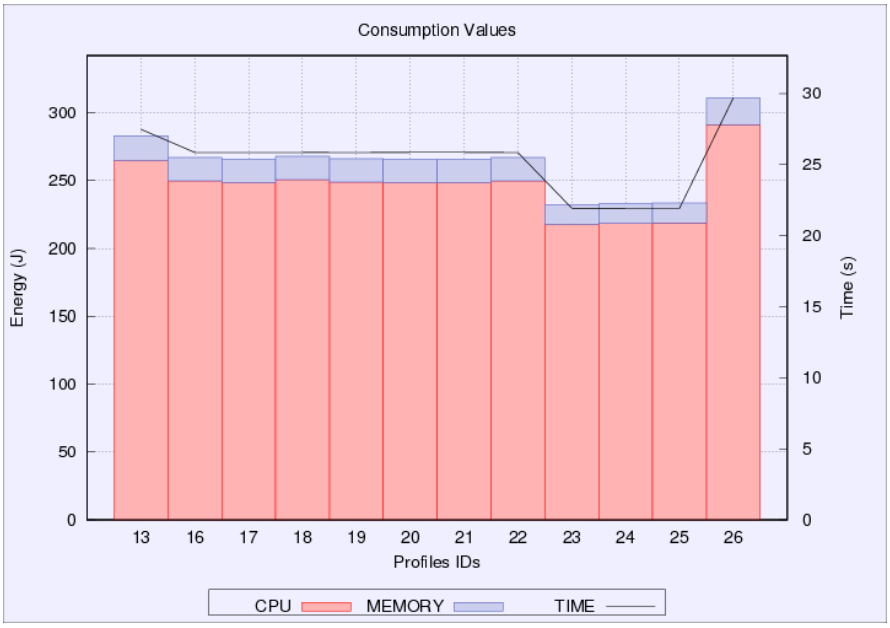


Figure 37.: Profiles without debug but with optimization options for fannkuch-redux.

this specific parameter allow to understand that the obtained values for all programs are completely equivalent.

Based on the examined behaviors, even given the impact the debug/profile options have on the resulting code (e.g. code size increasing), there is almost no influence considering

the measured aspects. Either using the comparison between profiles with non optimized categories or between profiles with the same optimization level, it is possible to verify that for all profiles there is little to none substantial impact regarding execution time and energy consumption. So, it is concluded that a user can perfectly opt for profiles with these characteristics without any kind of apprehension due to significant performance loss. If a project is in an initial state, *RelwithDebInfo* profiles are a good alternative considering the other usually chosen options.

In Section 5.5, some compilation profile sets were defined with the help of GCC and its categorization. From the performed research resulted nine categories that differentiate the profiles according to the purpose of the flags that compose them. It is convenient to reuse this study result and verify how the mentioned category options distinguish themselves based on the new aspects. During the current analysis, this approach allow to focus on more specific sets related to the compilation parameters. On charts contained in repository directory 2.6 are displayed some information regarding the results for all the programs using the considered categories. The categories and respective profiles that compose them are listed as follows:

1. Empty
   Profile ID: *1*;
2. Warning/Dialect Oriented
   Profile ID: *2*;
3. Debugging/Profiling without Warnings
   Profiles IDs: *3*, *4*, *6*;
4. Debugging with Warnings
   Profiles IDs: *5*, *7*, *8*, *9*, *10*, *11*, *12*;
5. Low Level Optimization
   Profile ID: *13*;
6. Optimization with Debug Information
   Profiles IDs: *14*, *15*;
7. Optimization Recommended Level
   Profiles IDs: *16*, *17*, *18*, *19*, *20*, *21*, *22*;
8. High Level Optimization
   Profiles IDs: *23*, *24*, *25*;
9. Code Size Optimization
   Profile ID: *26*;

Interpreting the results for each program, it is visible for the majority of the situations that no substantial differences are found in any categories. This conclusion is mostly because of two major causes.

On one hand, several used flags have little to no influence on the generated executable file. More specifically, it is verified that these options have very relevant tasks related to compilation process management (e.g. preference configuration, macros definition, dependency management), detection and transmission of anomalies (e.g. throw and management of warnings) and auxiliary information to code development (e.g. produce debuggable code, collecting profiling statistics). It is then possible to conclude that these options contain characteristics that assist the programmer during the development of software, and do not add up extra information on the generated code or, even if they do, the impact is residual. The presence of most of these options is indifferent to the produced executable, so the lack of impact on the final results may be considered normal.

On the other hand, the individual flags used don't have any influential effect on code based on the measured aspects. Due to high precision provided by the measurement framework, it possible to verify that in most cases the obtained differences are basically one-thousandth of a Joule or second. These values are sometimes so reduced that there is some difficulty to understand if it is related to the applied flag or just another system performance configuration. In most cases only in situations in which an optimization suite is applied (which may have at most between 43 and 102 flags) it is possible to witness some significant variation.

In addition, there are some selected options that are repeated throughout several tools. Almost half of the mentioned categories only contain one compilation profile, rendering impossible to perform any type of comparison.

Simultaneously considering all the profiles, it is naturally observed the same characteristics that were noticed before. They clearly allow to verify the proportion differences and possible nuances between multiple situations. Based on the presented profiles analysis made via charts (Figures 38-49), it is confirmed that, for most cases, the conclusions are precisely the same as before when approaching the program and tool viewpoint subject.

The more dynamic the program behavior is during the multiple profiles, the easier it is to prove the conclusions. Regarding the two programs with the most static behavior, it is difficult to make any absolute conclusion.

Using *thread-ring* it is impossible to conclude anything considering the used profiles. The produced results present random behaviors and, in some situations, even opposite response comparing with the rest of the programs. For several aspects, it is possible to notice that there are non optimized profiles containing more efficient results comparing

with optimized profiles, and some anomalies on proportions between factors based on the already observed ones.

After all, and in a very subtle way, in *regex-redux* case it is possible to observe some behaviors that are equally found on the rest of the programs. Such as small decreases in execution time and energy consumption upon comparing profiles with and without optimization, percentage differences between the most extreme examples on each aspect following the same pattern as the more dynamic programs, among others.

For the correlations that present a considerable order of magnitude, it is not relevant to indicate examples or any particular case since those are evident and recurrent occurrences. However, it proves interesting to identify some examples in which the execution time and memory energy consumption are correlated since these cases occur less frequently. Particularly between the profiles *11* and *12* for *k-nucleotide* and *chameneos-redux* it can be noted that both measures decrease even though the global energy consumption increases, or yet the opposite behavior shown between profiles *8* and *9* of *fasta*, and between *10* and *11* of *chameneos-redux*. Also, for the first 11 profiles of *spectral-norm* and *mandelbrot* both execution time and memory energy consumption values are practically stable.

Considering the complete set of profiles, no cases were found to be the most/least efficient regarding all of the contemplated programs. As previously mentioned, the generated binary depends on several factors beyond the options selected by the user. For instance, the characteristics of the source code are fundamental on defining which options, and to what extent, will effectively be applied by the compiler. Although some parameters might be part of the compilation profiles, in case the compiler considers that they are not adequate to the type of program being built, they may not be applied. Furthermore, the compiler assumes this and other decisions based on a series of heuristics towards an average system, and as such, there are no guarantees that better code will effectively be produced.

Nonetheless there are profile sets that provide more/less efficient results for the considered measures on most cases. Rankings in repository directory 2.2 classify all profiles for each analyzed program and the presented in Tables 20-23 (repository directory 2.1) consider the whole set of programs.

Examining particularly the ranking presented in Table 14 it is distinguished that, such as for the tools, there are no significant leaps in classification considering the various aspects. It can be also noted high similarity among the positions obtained by pairs in which a strong connection exists: total energy consumption - processor energy consumption and execution time - memory energy consumption. At last, it has been perceived that the most/least

Profile ID	Execution Time (s)	Total Energy (J)	CPU Energy (J)	Memory Energy (J)	Energy/Time (J/s)
1	20 (200)	23 (243)	21 (249)	19 (193)	19 (187)
2	21 (201)	20 (232)	18 (231)	18 (191)	18 (181)
3	25 (217)	22 (240)	20 (239)	22 (210)	13 (156)
4	17 (190)	18 (230)	18 (231)	16 (185)	15 (170)
5	15 (177)	20 (232)	18 (231)	14 (177)	22 (199)
6	24 (207)	25 (252)	22 (250)	21 (199)	21 (189)
7	23 (204)	21 (237)	19 (238)	20 (198)	17 (179)
8	18 (192)	17 (222)	17 (227)	17 (190)	15 (170)
9	18 (192)	16 (220)	15 (220)	18 (191)	11 (154)
10	16 (179)	15 (218)	14 (217)	15 (184)	16 (172)
11	22 (202)	19 (231)	16 (225)	21 (199)	13 (156)
12	19 (198)	24 (248)	22 (250)	21 (199)	20 (188)
13	12 (107)	13 (130)	12 (131)	11 (104)	10 (153)
14	8 (91)	5 (82)	4 (79)	8 (84)	1 (102)
15	7 (84)	8 (106)	7 (101)	7 (82)	8 (143)
16	3 (55)	6 (92)	6 (92)	3 (56)	9 (150)
17	5 (74)	4 (81)	4 (79)	5 (73)	5 (126)
18	13 (112)	11 (113)	10 (113)	12 (105)	4 (125)
19	10 (100)	7 (93)	5 (88)	10 (98)	3 (112)
20	11 (105)	12 (124)	11 (121)	12 (105)	12 (155)
21	9 (95)	10 (110)	9 (110)	9 (96)	6 (137)
22	6 (78)	9 (109)	8 (108)	6 (76)	11 (154)
23	2 (47)	1 (48)	1 (51)	2 (50)	3 (112)
24	4 (71)	3 (60)	3 (69)	4 (62)	2 (111)
25	1 (33)	2 (55)	2 (61)	1 (33)	7 (139)
26	14 (147)	14 (167)	13 (166)	13 (143)	14 (168)

Table 14.: Profiles ranked with 3 decimal points.

efficient option regarding total energy consumption and execution time does not always correspond to the best/worst option in terms of the ratio between these two components.

Comparing the gathered profiles, it is distinguished that *23*, *24* and *25* are the ones that present the most efficient results for the considered aspects in most cases. This characteristic becomes more evident the more rigorous the analyzed results are. On the other hand, the first 12 profiles are the ones that display the least efficient results in all the considered set. Even though for some subjects there are results that score slightly better among themselves within this group, upon comparing them with the generality of the profiles they remain the worst. Profiles *14* and *26* fit as the average case considering all the examined options. For all programs, these profiles show good balance for the several aspects, inclusively obtaining results closer to the best rather than the worst analyzed cases. The remaining 9 profiles (IDs ranging between *14*-*22*) also stand out as favourable, occupying the best ranking positions

immediately after the first considered group. While these elements do produce small differences in their results, they still occupy positions 4 to 12 of the ranking for the various subjects.

The several represented rankings contain position sequences with very close results for the considered measures. As these differences are grossly very reduced, it would be suitable to elaborate a ranking that would better illustrate the grouping of such equivalent values. By following the same approach used for the tool options, and through the ranking and significant charts, the following classifications for execution time, energy consumption, and both, are obtained:

Time Ranking:

1. *23, 25;*
2. *24;*
3. *16;*
4. *22;*
5. *17, 15;*
6. *18, 19, 20, 14;*
7. *21;*
8. *13;*
9. *26;*
10. *5, 10, 6, 12, 9, 1, 8, 4, 7, 11, 3, 2;*

Energy Ranking:

1. *23, 25, 24;*
2. *16, 17, 14, 19;*
3. *22, 15, 21, 18, 20;*
4. *13;*
5. *26;*
6. *9, 11;*
7. *7, 3, 8, 4, 10, 12, 1, 2, 6, 5;*

Time and Energy Ranking:

1. *23, 24, 25;*
2. *16;*

3. *17, 14, 22, 19, 15;*
4. *18, 20, 21;*
5. *13;*
6. *26;*
7. *9, 8, 10, 11, 7, 3;*
8. *1, 4, 5, 12, 2, 6;*

The presented abstractions allow to validate the observations mentioned along this subsection. Particularly, examining the classification that gathers the results for execution time and energy consumption, an excellent summary on how the considered profiles differentiate among themselves is obtained. This abstraction could inclusively have been more concise if some of the aspects had been examined with less rigour, and small nuances of the studied programs had been ignored.

5.6.5 *Parameters*

To conclude the discussion of the results gathered in this experimental study, it remains to analyze the desired points through the perspective of the compilation parameters. In particular, to verify which impact the flags (individually and together) have on the execution time and energy consumption of the programs, and also observe how they differ from each other. Some of these points have already been referenced throughout this study, but nevertheless, always in a more global perspective. The analysis will mainly rely on the same graphic and numerical material that was used in the previous subsection (Figures 38-49, Tables 20-23).

Most tools provide profiles that are composed of more than one compilation parameter. It is noted that only in 4 of the 26 compilation profiles this situation does not happen, and in only 1 case it is not a suite of flags. Therefore, the results obtained rarely concern individual options and only allow comparisons between sets. However, some of these parameters have a negligible impact on the measured points, which may lead to a reduction in the number of relevant options present per profile.

It is also noted that there is little diversity in the options provided. Only 26 different options are counted from a total of 144 parameters collected from the analyzed profiles. It is also observed that, excluding equivalent flags, optimization suits and options that were disregarded for the measurements made, there are only 15 parameters left, and the greater part is dedicated solely to the compilation stage.

This data allows us to conclude that the options provided are generally quite similar and focus mainly on the interaction with the user during the compilation process. Regarding

transformations in the generated code, the tools almost always opt for optimization suites (without adding any extra optimization flag) or including debug information.

In the set of parameters provided by the tools it is verified that not all of them perform transformations in the generated code. In particular, there are options that have the function of handling aspects related to messages that inform the user of possible anomalies detected during the compilation process. Also known as diagnostic messages options, it is verified that within the options gathered in this study there are seven parameters with these characteristics: *-fmessage-length=0*, *-fno-diagnostics-show-caret*, *-fshow-column*, *-W*, *-Wextra*, *-Wall*, and *-Wno-write-strings*. Through the analysis of the obtained results it emerges that these type of options do not have any influence on the measured aspects. It should be noted that in the presented methodology the compilation process is not measured and therefore any possible impact in it is not accounted for in the obtained results.

In this way and although they do not contribute to a greater efficiency in the generated executables, it is concluded that they are in fact a good choice by the tools due to the recognized importance of the category in question.

For the *-pipe* parameter, a behavior with the same characteristics is also verified. Being an option that tries to change the preference in the communication between build phases, it is noted that it also does not have any effect in the code generated.

Among the analyzed parameters there are also three options that do not produce any differentiating effect for the various cases considered. The first option is *fvisibility=default* which is observed to be declared explicitly as a parameter, but is already implicitly used by the compiler. The *-lm* option explicitly requires the linking of a library that will not be included if it is not required or otherwise will be requested in the same way for all analyzed cases (through the parameters required by the program itself). Finally, it is verified that the *-fomit-frame-pointer* option is explicitly declared but is used simultaneously with the suite *-O2* of which it is already part.

In this way it can be concluded for all the presented options that it is not possible to analyze their respective effect in the considered aspects. Although there is the possibility of producing modifications in the resulting program, it is verified that they will occur indistinguishably for all analyzed cases. It is also concluded from the presented reasons that the use of these options by the tools is redundant and therefore dispensable.

The conclusions are quite similar when considering parameters that alter the production of debuggable code or collect profiling statistics. Both types of options have in fact influence on the generated executable, such as making it have a larger size or the executed code

is spread over more pages. However, it is verified that they do not change the control flow of the code and therefore do not have any type of effect in the execution time or energy consumption of the program. It is also noted that this result extends to the various types of options used in this category regardless of their depth level, format variations or extension chosen. The great similarity in the results obtained for the several cases indicated also serves as proof of the mentioned facts. In this way it is concluded that the flags *-g*, *-g2*, *-g3*, *-ggdb*, *-ggdb3*, *-DNDEBUG* and *-pg* have no interference in the measured aspects.

Regarding the *-fPIC* option, diverse results are obtained depending on the type of program being considered. Although this parameter is not used in any case in an isolated way, it is observed that it is contained in five profiles (IDs *9*, *11*, *15*, *18* and *20*) which allows it to be compared with other similar cases. By analyzing the results obtained for the profiles in question throughout the various programs, it is found that they have different behavior in relation to the intended strands.

Compared with other equivalent profiles, they are sometimes found to be more efficient (e.g. *fannkuch-redux*, *fasta*, *meteor*), less efficient (e.g. *binary-trees*, *n-body*, *k-nucleotide*) or even equivalent (e.g. *spectral-norm*). However, although it is very subtle, it is observed that in most cases there is indeed some effect on the aspects measured. This data leads to the conclusion that the generation of position-independent code suitable for dynamic linking may in some cases in particular allow a greater efficiency in execution time and energy consumption. Nonetheless, the opposite behavior is also observed and therefore the use of this parameter must be done with some care in cases where such rigor is necessary.

The optimization levels clearly have the most efficient results for the elements analyzed. In addition to the default level (*-O0*), a further four sets (*-O1*, *-O2*, *-O3*, *-Os*) are used for different purposes. The flags they provide are either partially or fully contained in the remaining sets and can therefore be seen as extensions to each other. Depending on the suite and program considered, in total up to 43-102 individual flags can be enabled to control various aspects of the compilation process and resulting code. Finally, it is up to the compiler to decide based on assumptions and heuristics which options will actually be applied in code generation. The optimization attempt by the compiler has no guarantee that it will actually improve the resulting code, nor that the application of a given transformation will not detract from another aspect that might be relevant to another objective. In particular, we can even observe the application of the same set of optimizations for different levels of optimization.

Regarding the results obtained for the four relevant suites, it is concluded that in fact there is no option that is better or worse for all cases. It is noted that the classification of

the various options is highly dependent on the program concerned, and is sometimes very similar or even equivalent. Although not always very significant, it is verified that the level *O3* presents more efficient results for the great majority of the cases.

On the other hand, it is verified that the level *Os* is generally the one that has the least efficient results. This is mainly due to the nature of the suite itself and the objectives it seeks. In the particular case of *regex-redux* it is even observed that it has worse results in the analyzed strands than the default suite.

The remaining two levels have a very close behavior between them. Although the *O2* suite has more than double the options present in *O1*, it can be seen for the programs analyzed that only in the case of *fasta* there is a great difference between both. These results confirm that although the *O1* level is the most basic element of the cases considered, it is nevertheless quite competent in the considered aspects. Even so, there is a general rule that there is a slight rise in the *O2* level relatively to *O1*.

In order to be able to compare and quantify concretely the optimization levels used by the tools under study, five profiles representing the respective categories (IDs *1*, *13*, *19*, *25* and *26*) were selected. Table 15 shows the respective mean values obtained for the different strands considering all programs analyzed. It is also indicated the percent difference compared to the default level (*-O0*).

Optimization Level	Time (s)	Energy (J)	CPU (J)	Memory (J)	Energy/Time (J/s)
O0	16.862	181.611	169.872	11.739	10.582
O1	8.689 48.5%	85.511 52.9%	79.195 53.4%	6.316 46.2%	10.212 3.5%
O2	8.415 50.1%	82.182 54.7%	76.063 55.2%	6.119 47.9%	10.032 5.2%
O3	7.644 54.7%	74.058 59.2%	68.485 59.7%	5.573 52.5%	9.872 6.7%
Os	9.408 44.2%	93.879 48.3%	87.061 48.7%	6.818 41.9%	10.315 2.5%

Table 15.: Optimization levels results.

Analyzing the presented results, the same trends mentioned above are also observed. Clearly the aspect that stands out most is the large difference in values between the default level of optimization and the rest. Regardless of the level applied, mean gains between 40%-60% are obtained for the four measured elements. This significant difference shows once again that code optimization when compiling is a very efficient way to get improvements in the generated code without any extra effort by the programmer.

It is also noted that the level *Os*, although it has advantages in other areas, it does not allow such significant improvements compared to the other options (about 10% less than *O3*). The proximity between the *O2* and *O1* suites is also marked by the minimum percentage difference between the two (1.6%-1.8%) and is verifiable as well the slight ascendant of the superior level to the various strands.

The *O3* optimization suite made it possible to achieve improvements above 50% for the four components considered, and for the total energy consumption and processor it is even close to 60%. They are indeed quite impressive results proving the excellent performance of the compiler in this matter.

5.6.6 *Discussion*

The methodology described above involved, among other aspects, 12 programs, 18 development tools and 26 compilation profiles. The respective execution time, CPU and memory RAM energy consumption (individually and together) and the ratio between both factors (energy/time) were measured for each case. The results obtained from different perspectives that encompass the set of analyzed elements were presented and discussed.

From the point of view of the programs, it was verified through the numerical and graphical analysis that they have different characteristics with respect to the analyzed strands. In general, it was found that 2 programs (*thread-ring* and *regex-redux*) exhibit a very regular behaviour with small changes when compiled with different options, while the remaining ones show big behaviour changes throughout the various profiles. For these 10 programs was observed a great difference between the best and worst cases, mainly in *fasta* and *n-body*. *Meteor* and *fannkuch-redux* present respectively the smallest and largest results in terms of execution time, total energy consumption and processor energy consumption. In the case of memory energy consumption it was found that, with the exception of *reverse-complement*, the results obtained were generally more closely related.

From the data obtained it was also possible to testify some previous conclusions. In particular, it was verified the great impact of the CPU on energy consumption, being responsible for about 90% of the total value measured. It was also observed the great impact of the GCC optimizations flags in the programs the execution time and energy consumption (maximum improvements in the order of 86%). It was also concluded that, overall, the strands with the best and worst percentage improvements achieved are the processor (53%) and memory energy consumption (46%), respectively.

From the analysis made, it was possible to observe some correlations between the execution time and the different types of energy consumptions measured. In particular, it was concluded that, for the majority of the programs and profiles considered, there is a strong

link between the execution time and total energy consumption obtained. More specifically, it is verified that the memory energy consumption has quite influence in the program execution time (being largely responsible for the more abrupt changes). It was also concluded that, as a general rule, the ratio between energy consumed and execution time does not differ greatly from case to case.

From the perspective of the tools, it was observed that the majority of them give the user two compilation profiles with very clear and distinct objectives (debug and release). Through the data obtained for each studied profile, efficiency rankings were elaborated according to the execution time and energy consumptions for the respective tools. It was concluded that *Sphere Engine* and *GPS* are the best classified tools and on the other hand that *AWS Cloud9*, *Code::Blocks* and *Geany* stand out negatively. With regard to the BATs (CMake, qmake and Qbs) it is observed that in general are well classified in the various efficiency rankings and therefore prove to be an excellent option for the users.

Through the analysis between all the tools with similar profiles in terms of quantity and objectives, it was observed however that the differences obtained in the mentioned strands are not very relevant for most cases.

The tools were also classified considering only the results obtained for the profile that they provide to the user by default. It was concluded that except for the *Sphere Engine*, all tools have relatively close results in terms of execution time and energy consumption.

The analysis of the profiles was initially made using categories in which they were grouped according to objectives and similar characteristics. It is concluded that, except for categories with optimization suites, there are no significant differences between the values obtained for the analyzed profiles. It was also verified that the majority of the parameters are used for the configuration of the compilation process, management of diagnostic messages or to produce information of debugging and profiling. It was also proved that those parameters do not have a significant impact on the execution time and energy consumption of the programs.

In the case of profiles that include optimizations, it was again confirmed the great influence they have in the execution time and energy consumption. The analysis by categories also allowed to observe that profiles that include optimizations of a moderate level together with options related to debugging, are in fact good choices for the developer.

Analyzing for each program the set of profiles as a whole, it was also found that for the majority of cases there are no substantive differences between non-optimized profiles. Through the same analysis, the previously observed correlations for the elements and strands under study were also confirmed.

Following the previous approaches and assumptions, rankings were also developed for the analyzed profiles. It was found that there are no profiles that are the best or worst for all programs. However, it has been found that for the majority of cases the most efficient sets are: profiles *23* to *25*; then between *14* and *22*; then the profiles *14* and *26*; and finally the first 12 options.

Finally, we analyzed the results from the point of view of the compilation parameters, concluding that the majority have little to no impact on the way the program is executed. It was found that there is a large set of options that have other objectives relevant to the compilation process (e.g. management, warnings, debug). However, they do not change the control flow of the code and therefore no impact on the execution time and energy consumption of the program.

Lastly, the four levels of optimization were compared taking into account the measured strands. It was concluded that for most programs the *O3* level produces more efficient results, with a mean gain of 56% for all aspects by comparison to the default level. Then, the levels *O2* (51.6%) and *O1* (49.8%) appear with very similar values. Finally, but still with very interesting values, appears the level *Os* (45.3%). This data allows us to conclude again that optimizations are a very fast and effective way for programmers to improve the energy efficiency of their programs.

5.7 SUMMARY

In this chapter, an experimental thematic study was presented that had as one of the main objectives to investigate from an energy perspective the performance of the programs generated by IDEs.

The study begins with the analysis of some of the main features of Integrated Development Environments. A broad research was carried out to ascertain for example the origin of these tools, their advantages and disadvantages with regard to the existing alternatives or their differentiating factors. It has been proven that in fact IDEs are essential programming tools to increase the productivity of any type of developer. Whatever the format, features and functionalities they present, they are clearly a key development model today.

Subsequently a research was carried out in the market of this type of tools and selected those that respected some criteria that we consider pertinent. In the course of this process it was found that because they have similar characteristics considering compilation strategy, it would also be interesting to analyze tools with Build Automation Tools features. In total, 18 tools were selected that passed the filtering process of a set of 49 explored options. During

this process it was also possible to perform a comparative study between the selected tools and to verify for example some particularities and similar/divergent aspects between the various options.

After gathering the necessary tools, a study was carried out to find out the type of strategy used by the IDEs to generate code automatically. It was concluded that they mainly use compilation profiles that allow the user to compile programs according to a set of predefined parameters. It was verified that they use three different ways of obtaining this mechanism (not optimized, own profiles and BATs profiles) and also the main characteristics of each one were analyzed.

Next, the compilation profiles of the tools under study were collected, resulting in 51 elements (29 distinct). Through this set, it was concretely ascertained the intended objectives of the tools for each option provided. It was also possible to survey some data that allowed to establish some comparisons and considerations between the various tools.

In order to perform a more detailed analysis of the parameters present in the compilation profiles, it was necessary to review and deepen some topics related to the compiler and programming language used. In particular, the various steps of the GCC compilation process for C programs have been described and what kind of options are allowed during the same. Finally, 16 categories of parameters considered by GCC were presented together with the detailing of demonstrative examples for each one.

After gathering the desired compilation profiles and analyzing the stance of GCC relative to the compilation parameters, it was then possible to examine the options contained in the profiles obtained. It was verified that in total, 144 parameters (28 distinct) are used and that they are inserted in nine different categories according to GCC. For each of these elements a description and contextualization was then performed according to the respective categories to which they belong. With this data it was possible to verify in detail the respective effect that each one of the options intends to perform during the compilation. Taking into account some data collected, it was found that it was possible to group the options obtained in a more summarized way into four sets according to their function: control of the compilation process, management of diagnostic messages, provision of debug/profile information and optimization of the generated code.

Following the same approach, the compilation profiles were analyzed according to the characteristics of their parameters. It was concluded that it is possible to group them into nine distinct categories and that the vast majority use debugging and optimization options. This new data made it possible to perform some more comparisons between the various

tools from the perspective of the profiles and their compilation parameters.

After researching the elements relevant for the experimental study, it was necessary to investigate the elements that allowed the respective analysis. It was verified that the target machine and the measurement framework used in the previous experimental study continued to be perfectly suited to the objectives intended here. In this way it was decided to reuse them.

It was also necessary to investigate which benchmarks allowed to achieve the desired results. Very strict search criteria were defined in order to find solutions not only suitable for study but also with considerable relevance for the community. In particular, it was intended that they were sophisticated enough to be challenging for both the hardware and the compiler and even if they were based on problems with different scopes and backgrounds in order to promote the diversity of the resources used. The choice fell on the Computer Language Benchmarks Game and were selected the 12 submissions in C best positioned in the project performance ranking.

The applied experimental methodology was also inspired by the study presented in the previous chapter. After selecting the desired elements and filtering the cases with redundant behavior, the process of measurement of execution time and energy consumptions for all the programs, profiles and intended tools was performed. The obtained results were then processed and presented in more appealing and relevant formats for analysis: charts, tables, HTML pages, and rankings.

Based on this data, the results were presented and discussed through different perspectives regarding the elements considered. In addition to the efficiency of the measured aspects, other aspects relevant to the developers in general were also analyzed.

It was observed that the analyzed programs have different behaviors in relation to the measured aspects. In particular, it was found that there is a very strong connection between execution time-total energy consumption, between total energy consumption-processor and also between execution time-memory energy consumption.

Regarding the efficiency level of the tools it has been noted that there are significantly different results among the respective profile sets. It has been verified that the majority opts by providing 2 compilation profiles with very distinct characteristics (*Debug* and *Release*) and there does not exist significant difference between these tools concerning the obtained results. It has been yet noted that the Build Automation Tools present fairly interesting results and prove to be an excellent option for users. The results also allowed the elaboration of a ranking for the tools concerning the given aspects.

Regarding the obtained results for the compilation profiles it was verified that the results are very similar, inclusively for all profiles that do not include optimization suites. Individually comparing each profile it was possible to establish a ranking according to the examined strands, and state which options are the most/least efficient.

Ultimately the results obtained for each individual parameter were analyzed. It was concluded that the vast majority of options did not influence significantly the considered aspects. As for optimization levels it has been noted that all display very significant improvements (40%-60%) for the measured subjects. This information corroborate the conclusion that optimizations are, indeed, a fast, simple and effective way to obtain improvements for the energy consumption of the generated code.

This study was carried out successfully and allowed to obtain and deepen a vast set of information. It was possible to inspect the most successful types of programming tools, namely the different existing varieties and which strategies they use to automatically generate code. It was yet viable to enrich the previously started research concerning compilers, and with particular focus on GCC. It was possible to analyze and describe a large group of compilation parameters and classify them according to their purposes, as well as use a diversified and optimized set of benchmarks. A methodology that encompassed such diverse elements was successfully designed and enabled the desired results to be achieved viably and efficiently. Through a pertinent processing of the results, it was possible to obtain more easily the intended answers for this study and still to establish some interesting observations regarding aspects directly related to the developer.

6

CONCLUSION

In this work have been examined some problems and limitations that arose in the IT industry (as a result of rapid development as well as of the high demands) such as increasing energy consumption and the negative impact on the environment. These problems reveal the need for a change in the production paradigm throughout the IT industry, in order to reverse this negative trend. In this context, was presented the Green Computing concept which means the study and practice of efficient and environmentally sustainable use of computers and related resources. Use new energy-generation techniques, reduce the use of hazardous materials and decrease pollution and the environmental impact were some of the objectives of the paradigm described along the report. After a historical review of a few landmarks and their evolution, was explained the importance of Green Computing both today and in the future. Some of the good practices and technical approaches currently in use were presented; for example Cloud Computing, Power Optimization, Virtualization and Grid Computing were analyzed.

Afterwards, microprocessors were analyzed from the perspective of Green Computing. More specifically, was exemplified their heterogeneous utilization nowadays, the impact on the energy consumption of a computer system and the increasing technological importance of microprocessors. From that research it became clear that optimization during code generation is a crucial way to combat excessive energy consumption. It was possible to conclude from the analysis of the evolution that manufacturers and consumers of products with microprocessors require increasingly more energy efficient solutions.

Taking into consideration the importance of microprocessors and their impact on energy consumption in a computer system, the modern compiler design was analyzed with the goal to understand which techniques already used have potential to improve the energy efficiency of the generated code. This search made possible to conclude that although the low energy consumption are still not a priority on the compilation process, it is already considered as one of the four optimal properties of the produced code. In the set of all optimization techniques applied with energy consumption impact, there are some evident

ones in efficiency order: making the execution of the program faster, lowering the CPU voltage, reduce the amount of bit switching and specific domain-dependent methods. Just-In-Time compilation is also other approach that bring some advantages on the generated code allowing a significant improvement on the overall energy consumption.

After the theoretic investigation, two experimental studies were done with the objective of applying in some examples some of the acquired knowledge and check, in practice, what is the real impact.

The covered topics of the first study are according with multiple aspects of the study on modern compilers like for example the existing correlation between the execution time and program energy consumption. Specifically, it focuses on analyzing and comparing the impact of the multiple levels of GCC optimization on the C, C++, Go and Objective-C programs consumption. The results were obtained via a measuring tool developed by the team of this project, that allows, among many other functionalities, to perform analysis of Machine-Specific Registers containing information about the energy consumed by the CPU, RAM and GPU during a given period of time. After all the necessary elements (testing platform, measurement software and measured software) are defined and gathered, a study methodology was planned with the goal to include all the mentioned aspects around the final objective and, in the end, a presentation and final results discussion happened.

With this experimental study, it was possible to conclude that the GCC optimization have a great impact on the energy consumption using the already mentioned programming languages. We found too that for the majority of the cases -Ofast level is the most efficient unlike -O1 and -Os options. It was verified too that a correlation between execution time and program energy consumption globally exist considering the hardware elements and programming languages. All this information indicate clearly that if a software developer wants, without effort, to reduce its software consumption, he only needs to apply on the sophisticated optimization levels that the compiler offers to achieve it. Besides the obtained results, this study was very important because it made possible not only to create a new measuring framework but to gain experience in a lot of important aspects that allowed to get into more and deeper analysis details.

The results of this study were also presented at the *2015 IEEE 13th International Scientific Conference on Informatics*[1]. We were invited to carry out an extension to the initial work to be published in *Journal Acta electrotechnica et Informatica No. 1, Vol. 16, 2016*[2].

The second experimental study aimed to analyze a specific tool type that was essential to developers in their daily basis and verify what was the real energy impact on the pro-

1 https://ieeexplore.ieee.org/document/7377807
2 http://www.aei.tuke.sk/papers/2016/1/2016-1.htm#BRANCO

duced code. Study IDEs and some BATs was the way to go, mainly understanding what are the differences between the compilation profiles provided to users based on the energy efficiency. It was also possible to deepen the research already begun on the compilers and their compilation parameters. At the same time, the CLBG (a project that gathers and compares software challenge solutions in multiple programming languages) properties were studied and the their use as a benchmark source for the main objective were confirmed.

After completing the research phase, it was possible to gather all the new necessary elements, to define the methodology to be applied and to process the obtained results to a more adequate format to the main analysis.

Through the discussion of the results it was possible to again verify that there is a correlation between execution time and energy consumption of the programs. In particular, it is verified that the execution time is strongly related to the energy consumption of the memory, although it is observed that the processor is responsible for most of the expense.

Regarding the tools analyzed, it was verified that there is a great similarity between the options they have. Usually they choose to provide the user with two profiles: one with appropriate options for an early software development phase (e.g., warnings, debug/profile options) and a more suitable one for the product release phase (e.g. sophisticated optimizations). However, with the exception of options with different levels of optimization, the results obtained are quite similar. Generally, it has been found that BATs offer more efficient options for the analyzed aspects.

Regarding the compilation parameters, it was verified that the majority has no impact on the execution time and the program's energy consumption. Most of the options do not change the flow control of the code, aiming at other types of tasks such as: configuration of the compilation process, management of diagnostic messages or to produce debugging and profiling information. This behavior leads to the conclusion that, except in situations where such a requirement exists, the programmer can choose without much concern the use of several individual compile flags without loss relevant to the energy consumption of his code. Regarding the optimization levels used by the tools under study, it is verified that levels 3 and 2 are generally the ones that obtain greater reductions in the energy consumption of the programs (on average 57 %).

It was also possible to elaborate some rankings of the studied tools taking into account several characteristics of both the tools themselves and the executables they produce. For the profiles and compilation parameters a similar analysis was carried out and the results obtained were also equivalent. This study proved to be very opportune as it enabled us to solidify acquired knowledge, to deepen a topic that is very relevant to any developer, and to obtain results with very interesting indicators.

The various studies have also enabled us to gather a huge set of information that can also be used as a good workbench for other green oriented research.

BIBLIOGRAPHY

AEEC. The Energy Conservation Center, Japan, 2015. URL http://www.asiaeec-col.eccj.or.jp/contents01.html.

Christopher Barnatt. ExplainingComputers - Green Computing, 2012. URL http://explainingcomputers.com/green.html.

L. Belinda. Measuring application power consumption on the Linux operating system, 2015. URL https://software.intel.com/en-us/blogs/2013/06/18/measuring-application-power-consumption-on-linux-operating-system.

Karen Bemowski. Windows ITPro - Power Management Software for Windows Workstations, 2010. URL http://windowsitpro.com/windows/buyers-guide-power-management-software-windows-workstations.

Michael R Betker, John S Fernando, and Shaun P Whalen. The History of the Microprocessor. *Bell Labs Technical Journal*, pages 29–56, 1997. ISSN 10897089. doi: 10.1002/bltj.2082.

Stefan Brunthaler. Inline caching meets quickening. In Theo D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, pages 429–451, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14107-2.

L N Chakrapani, P Korkmaz, V J Mooney III, K V Palem, K Puttaswamy, and W F Wong. The Emerging Power Crisis in Embedded Processors: What Can a Poor Compiler Do? *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 176–180, 2001. doi: 10.1145/502217.502246. URL http://doi.acm.org/10.1145/502217.502246.

Premangshu Chanda, Subrata Chanda, Pallab Kanti Mukherjee, Shalabh Agarwal, and Asoke Nath. Scope and issues in green compiler. *International Research Journal of Computer Science(IRJCS) : ISSN: 2393-9842*, 3:86–93, 01 2017.

Jee Whan Choi, D. Bedard, R. Fowler, and R. Vuduc. A roofline model of energy. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 661–672, May 2013. doi: 10.1109/IPDPS.2013.77.

CMake Reference Documentatiom. Cmake reference documentation, 2018. URL https://cmake.org/cmake/help/v3.11/.

CNS. Computer Science Department The University of Texas at Austin - Toward Lowering the Power Consumption of Microprocessors, 2012. URL https://www.cs.utexas.edu/news-events/news/2012/toward-lowering-power-consumption-microprocessors.

Computer Weekly. Do you run a green machine?, 2006. URL http://www.computerweekly.com/feature/Do-you-run-a-green-machine.

Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, and João Saraiva. Towards a green ranking for programming languages. In *Proceedings of the 21st Brazilian Symposium on Programming Languages*, page 7. ACM, 2017.

Jem Davies. Partnerships, Standards and the ARM GPU Perspective. In *Compute Power with Energy Efficiency*, page 45, 2012.

Jay Dietrich, Roger Schmidt, Mike Hogan, and Gerry Allen. The green data center. Technical Report May, IBM, 2008.

Daniel Eran Dilger, 2016. URL http://appleinsider.com/articles/14/06/04/apples-top-secret-swift-language-grew-from-work-to-sustain-objective-c-which-it-now-a

The Economist. Pursuing sustainability, 2008. URL http://www.economist.com/debate/sponsor/136.

Brad Ellison. The Problem of Power Consumption in Servers. *Energy Efficiency for Information Technology*, pages 1–17, 2009.

Calle Erlandsson. The four stages of compiling a c programming, 2018. URL https://www.calleerlandsson.com/the-four-stages-of-compiling-a-c-program/.

Faiza Fakhar, Owais Malik, et al. Distributed green compiler. In *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, pages 421–426. IEEE Computer Society, 2011.

Faiza Fakhar, Barkha Javed, Raihan ur Rasool, Owais Malik, and Khurram Zulfiqar. Software level green computing for large scale systems. *Journal of Cloud Computing: Advances, Systems and Applications*, 1(1):4, May 2012. ISSN 2192-113X. doi: 10.1186/2192-113X-1-4. URL https://doi.org/10.1186/2192-113X-1-4.

William Forrest, James M Kaplan, and Noah Kindler. Data centers: How to cut carbon emissions and costs. *McKinsey on business technology*, 14(6), 2008.

Ian Foster. What is the grid? A three point checklist, 2002.

GCC team. Using the GNU Compiler Collection (GCC): Optimize Options, 2014. URL http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

GCC team. GCC 4.5 Release Series, 2016a. URL https://gcc.gnu.org/gcc-4.5/.

GCC team. Java and GNU, 2016b. URL https://gcc.gnu.org/java/.

GCC team. Programming Languages Supported by GCC, 2016c. URL https://gcc.gnu.org/onlinedocs/gcc/G_002b_002b-and-GCC.html.

GCC team. Status of Supported Architectures from Maintainers' Point of View, 2017. URL https://gcc.gnu.org/backends.html.

GCC team. Using the gnu compiler collection (gcc): Inline, 2018a. URL https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc/Inline.html.

GCC team. Using the gnu compiler collection (gcc): Invoking gcc, 2018b. URL https://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html#Invoking-GCC.

Prodromos Gerakios, Nikolaos Papaspyrou, and Konstantinos Sagonas. Race-free and memory-safe multithreading: design and implementation in cyclone. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, pages 15–26. ACM, 2010.

Sarah Gingichashvili. The Future of Things - Green Computing, 2007. URL http://thefutureofthings.com/3083-green-computing/.

Isaac Gouy. Computer language benchmarks game wiki, 2012. URL http://wiki.c2.com/?ComputerLanguageBenchmarksGame.

Isaac Gouy. Alioth: The computer language benchmarks game: Project home, 2018a. URL https://alioth.debian.org/projects/benchmarksgame/.

Isaac Gouy. The computer language benchmarks game, 2018b. URL https://benchmarksgame.alioth.debian.org/.

Sam Grier. IT Managers Inbox - PC Power Management Solutions, 2009. URL http://itmanagersinbox.com/1399/pc-power-management-solutions.

Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. Springer Publishing Company, Incorporated, 2nd edition, 2012. ISBN 1461446988, 9781461446989.

Tarik Guelzim and Mohammad S Obaidat. *Handbook of Green Information and Communication Systems*, chapter Chapter 8, pages 209–227. Academic Press, 2013.

Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using RAPL. *ACM SIGMETRICS Performance Evaluation Review*, 40, 2012.

R R Harmon and N Auseklis. Sustainable IT services: Assessing the impact of green computing practices. In *Management of Engineering Technology, 2009. PICMET 2009. Portland International Conference on*, pages 1707–1717, 2009. doi: 10.1109/PICMET.2009.5261969.

Paul Johannes Mattheus Havinga. Design techniques for energy efficient and low-power systems. *Mobile multimedia systems*, pages 2.1—-2.52, 2000.

Jackson He. Datacenter Power Management: Power Consumption Trend, 2008. URL https://communities.intel.com/community/itpeernetwork/datastack/blog/2008/02/20/datacenter-power-management-power-consumption-trend.

J. Henkel and S. Parameswaran. *Designing Embedded Processors: A Low Power Perspective*. Springer Publishing Company, Incorporated, 1st edition, 2007. ISBN 1402058683, 9781402058684.

Andrei Homescu and Alex Şuhan. Happyjit: a tracing jit compiler for php. *ACM SIGPLAN Notices*, 47(2):25–36, 2012.

Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *ACM SIGPLAN Notices*, volume 38, pages 38–48. ACM, 5 2003.

IBM Staff. Green IT – Energy Efficiency, 2015. URL http://www-03.ibm.com/systems/z/advantages/energy/.

Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, 325462-053 edition, September 2015.

Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 325462-064us edition, October 2017.

Intel Staff. 2013 Corporate Responsibility Report. Technical report, Intel, 2013. URL http://csrreportbuilder.intel.com/PDFFiles/CSR_2013_Full-Report.pdf.

Intel Staff. Intel Sustainability Initiatives and Policies, 2015a. URL http://www.intel.com/content/www/us/en/corporate-responsibility/sustainability-initiatives-and-policies.html.

Intel Staff. Intel 64 and IA-32 Architectures Software Developer Manuals, 2015b. URL http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

ITCandor. Microprocessors – Intel Leads, 2012. URL http://www.itcandor.com/microprocessor-q312/.

ITCandor. Microprocessor Market Down, 2013. URL http://www.itcandor.com/chip-q213/.

Gaurav Jindal and Manisha Gupta. Green Computing "Future of Computers". *International Journal of Emerging Research in Management & Technology*, pages 14–18, 2012.

Dong-Heon Jung, Soo-Mook Moon, and Sung-Hwan Bae. Evaluation of a java ahead-of-time compiler for embedded systems. *The Computer Journal*, 55(2):232–252, 2011.

Mahmut Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Wu Ye. Influence of compiler optimizations on system power. In *Proceedings of the 37th Annual Design Automation Conference*, pages 304–307. ACM, 2000.

Mahmut Kandemir, N Vijaykrishnan, and Mary Jane Irwin. Compiler optimizations for low power systems. In *Power aware computing*, pages 191–210. Springer, 2002.

Rajesh K Karmani, Amin Shali, and Gul Agha. Actor frameworks for the jvm platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20. ACM, 2009.

Ian King. Intel Forecast Shows Rising Server Demand, PC Share Gains, 2015. URL http://www.bloomberg.com/news/articles/2015-07-15/intel-forecast-shows-server-demands-makes-up-for-pc-market-woes.

Jonathan Koomey. Growth in data center electricity use 2005 to 2010. *Analytical Press*, 2011.

W Parmer Lane and Lizy K John. Impact of Virtual Execution Environments on Processor Energy Consumption and Hardware Adaptation. *Computing Systems*, pages 100–110, 2006. doi: 10.1145/1134760.1134775.

C. Lengauer. *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*. Lecture Notes in Computer Science. Springer, 2004. ISBN 9783540221197. URL https://books.google.es/books?id=dZokVMDMKl8C.

Wing Hang Li, David R White, and Jeremy Singer. Jvm-hosted languages: they talk the talk, but do they walk the walk? In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 101–112. ACM, 2013.

Luís Gabriel Lima, Francisco Soares-Neto, Paulo Lieuthier, Fernando Castor, Gilberto Melfe, and João Paulo Fernandes. Haskell in green land: Analyzing the energy behavior of a purely functional language. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 517–528. IEEE, 2016.

R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert. Hipacc: A domain-specific language and compiler for image processing. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):210–224, Jan 2016. ISSN 1045-9219. doi: 10.1109/TPDS.2015.2394802.

Lauri Minas and Brad Ellison. *Energy efficiency for information technology: How to reduce power consumption in servers and data centers*. Intel Press, 2009. ISBN 1-934053-20-1.

Christopher Mines. GreenBiz - 4 Reasons Why Cloud Computing is Also a Green Solution, 2011. URL http://www.greenbiz.com/blog/2011/07/27/4-reasons-why-cloud-computing-also-green-solution.

Scott Mueller. *Upgrading and Repairing PCs (17th Edition)*. Que Corp., Indianapolis, IN, USA, 2006. ISBN 0789734044.

R. Sekar N. Hasabnis, R. Qiao. Checking correctness of code generator architecture specifications. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO) 2015*, pages 167–178, Feb 2015. doi: 10.1109/CGO.2015.7054197.

Newsweek Staff. Newsweek Green Rankings 2011: Full Methodology, 2011. URL http://www.newsweek.com/newsweek-green-rankings-2011-full-methodology-68315.

Wellington Oliveira, Renato Oliveira, and Fernando Castor. A study on the energy consumption of android app development approaches. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 42–52. IEEE, 2017.

Oracle team. Oracle developer studio ide - oracle® developer studio 12.5: Overview, 2018. URL https://docs.oracle.com/cd/E60778_01/html/E60744/gkofj.html.

James Pallister, Simon J Hollis, and Jeremy Bennett. Identifying compiler options to minimize energy consumption for embedded platforms. *The Computer Journal*, 58(1):95–109, 2013.

Zhelong Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 12 pp.–, March 2006a. doi: 10.1109/CGO.2006.38.

Zhelong Pan and Rudolf Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 319–332, Washington, DC, USA, 2006b. IEEE Computer Society. ISBN 0-7695-2499-0. doi: 10.1109/CGO.2006.38. URL http://dx.doi.org/10.1109/CGO.2006.38.

Mark Papermaster. The Future of Energy Efficient Information Technology. Technical report, AMD, 2014.

Amisha Parikh, Soontae Kim, M Kandemir, Narayanan Vijaykrishnan, and Mary Jane Irwin. Instruction scheduling for low power. *Journal of VLSI signal processing systems for signal, image and video technology*, 37(1):129–149, 2004.

Andy Patrizio. The history of visual development environments, 2013. URL https://www.mendix.com/blog/the-history-of-visual-development-environments-imagine-theres-no-ides-its-difficult-i

Tomasz Patyk, Harri Hannula, Pertti Kellomaki, and Jarmo Takala. Energy consumption reduction by automatic selection of compiler options. *2009 International Symposium on Signals, Circuits and Systems*, pages 1–4, 2009. doi: 10.1109/ISSCS.2009.5206106. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5206106{&}contentType=Conference+Publications.

Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, pages 256–267, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5525-4. doi: 10.1145/3136014.3136031. URL http://doi.acm.org/10.1145/3136014.3136031.

Sviatoslav Pestov, Daniel Ehrenberg, and Joe Groff. Factor: A dynamic stack-based programming language. In *Acm Sigplan Notices*, volume 45, pages 43–58. ACM, 12 2010.

Qbs Manual. Qbs manual, 2018. URL http://doc.qt.io/qbs/index.html.

qmake Manual. qmake manual, 2018. URL http://doc.qt.io/qt-5/qmake-manual.html.

Kaushik Roy and Mark C Johnson. Software design for low power. In *Low power design in deep submicron electronics*, pages 433–460. Springer, 1997.

Sanghita Roy and Manigrib Bag. Green Computing-New Horizon of Energy Efficiency and E-Waste Minimization–World Perspective vis-à-vis Indian Scenario. *CSI, India*, pages 64–69, 2009. URL http://ww.w.csi-sigegov.org/emerging_pdf/8_64-69.pdf.

Ryan H. Intel, AMD & ARM Processors, 2012. URL https://kb.wisc.edu/page.php?id=4927#benchmarks.

Joshua Saddler. GCC optimization - Gentoo Wiki, 2016. URL https://wiki.gentoo.org/wiki/GCC_optimization.

Joshua Saddler. Gcc optimization - gentoo wiki, 2018. URL https://wiki.gentoo.org/wiki/GCC_optimization.

Hendra Saputra, Mahmut Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, Jie S Hu, Chung-Hsing Hsu, and Ulrich Kremer. Energy-conscious compilation based on voltage scaling. In *ACM SIGPLAN Notices*, volume 37, pages 2–11. ACM, 7 2002.

Aibek Sarimbekov, Andrej Podzimek, Lubomir Bulej, Yudi Zheng, Nathan Ricci, and Walter Binder. Characteristics of dynamic jvm languages. In *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, pages 11–20. ACM, 2013.

Jun Shirako, David M Peixotto, Vivek Sarkar, and William N Scherer. Phaser accumulators: A new reduction construct for dynamic parallelism. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. Optimization coaching: optimizers learn to communicate with programmers. In *ACM SIGPLAN Notices*, volume 47, pages 163–178. ACM, 10 2012.

Staff AMD. 2013/2014 Corporate Responsibility Report. Technical report, AMD, 2014. URL http://www.amd.com/Documents/AMD-2013-2014-CR-Report.pdf.

Jonathan Strickland. How Shared Computing Works - HowStuffWorks, 2008. URL http://computer.howstuffworks.com/shared-computing.htm.

Zhenyu Tang, N. Chang, Shen Lin, Weize Xie, S. Nakagawa, and Lei He. Instruction prediction for step power reduction. In *Proceedings of the IEEE 2001. 2nd International Symposium on Quality Electronic Design*, pages 211–216, 2001. doi: 10.1109/ISQED.2001.915229.

The World Bank. CO2 emissions (kt), 2014. URL http://data.worldbank.org/indicator/EN.ATM.CO2E.KT/countries?order=wbapi_data_value_2010+wbapi_data_value+wbapi_data_value-first&sort=desc.

Sebastian Thiel. A git mirror of the benchmarksgame cvs repository, 2018. URL https://github.com/Byron/benchmarksgame-cvs-mirror.

TIOBE Index. Tiobe index, 2018. URL https://www.tiobe.com/tiobe-index/.

Tirias Research. AMD Targets Accelerating Energy-Efficiency Gains. Technical report, Tirias Research, 2014. URL http://www.amd.com/Documents/Tirias-Research-Whitepaper.pdf.

Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, 1994.

Nick Touran. What is nuclear? / power basics – the difference between power and energy, 2017. URL https://whatisnuclear.com/physics/power_basics.html.

Madhavi Valluri and Lizy John. Is compiling for performance== compiling for power. *Interaction between compilers and computer architectures*, page 101, 2001. URL http://books.google.com/books?hl=en{&}amp;lr={&}amp;id=pE4guNvx8u4C{&}amp;oi=fnd{&}amp;pg=PA101{&}amp;dq=Is+Compiling+for+Performance+==+Compiling+for+Power?{&}amp;ots=Gf3qnsc59G{&}amp;sig=tW07EpiJySGmojXzJ4HsjotGPWQ.

Madhavi Valluri, Lizy John, and Heather Hanson. Exploiting compiler-generated schedules for energy savings in high-performance processors. In *Proceedings of the 2003 international symposium on Low power electronics and design*, pages 414–419. ACM, 2003.

Veracode. Integrated development environment ides — veracode, 2018. URL https://www.veracode.com/security/integrated-development-environments.

William Von Hagen. *The definitive guide to GCC*. Apress, 2011.

Kevin Williams, Jason McCandless, and David Gregg. Dynamic interpretation for dynamic scripting languages. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 278–287. ACM, 2010.

Carey Wodehouse. Compiled, interpreted languages, and jit compilers explained, 2017. URL https://www.upwork.com/hiring/development/the-basics-of-compiled-languages-interpreted-languages-and-just-in-time-compilers/.

Michael Wong. Let's Build a Smarter Planet. In *IBM Green Computing*, page 32, 2010.

Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *ACM Sigplan Notices*, volume 45, pages 377–388. ACM, 1 2010.

Han-Saem Yun and Jihong Kim. Power-aware modulo scheduling for high-performance vliw processors. In *Low Power Electronics and Design, International Symposium on, 2001.*, pages 40–45, 2001a. doi: 10.1109/LPE.2001.945369.

Han-Saem Yun and Jihong Kim. Power-aware modulo scheduling for high-performance vliw processors. In *Low Power Electronics and Design, International Symposium on, 2001.*, pages 40–45. IEEE, 2001b.

Yan Zhang and Nirwan Ansari. *Handbook of Green Information and Communication Systems*, chapter Chapter 12, pages 331–352. Academic Press, 2013.

A

SUPPORT MATERIAL

From the information processed regarding the various elements analyzed throughout the study presented in Section 5.6, resulted a large volume of data (more than 100 rankings and also more than 1000 charts, tables and HTML pages). This fact made it impossible to expose all the data obtained in the discussion presented or even in this document. It is, therefore, intended that this appendix act as an auxiliary tool in the analysis carried out, containing some of the remaining data considered more relevant for the mentioned study. The set with all the processed information can be consulted in the repository of this study[1] and in the project website[2].

In this appendix, a total of 24 charts and 8 rankings are presented referring to the values obtained by the 26 profiles measured for the 12 programs (Figures 38-49 and Tables 16-19) and 18 tools (Figures 50-61 and Tables 20-23) analyzed. In either case, the data portrays information regarding the 4 measured strands: execution time and energy consumption of the processor and memory (individually and together). In the particular case of the rankings, the ratio between the total energy consumed and the execution time is also indicated. In the Section 5.6 it is explained in more detail how all these values were obtained, what is the intended objective with its analysis and which conclusions can be drawn.

1 https://github.com/david-branco/programmingtoolsenergyconsumption
2 www.di.uminho.pt/~gepl/OCGREC

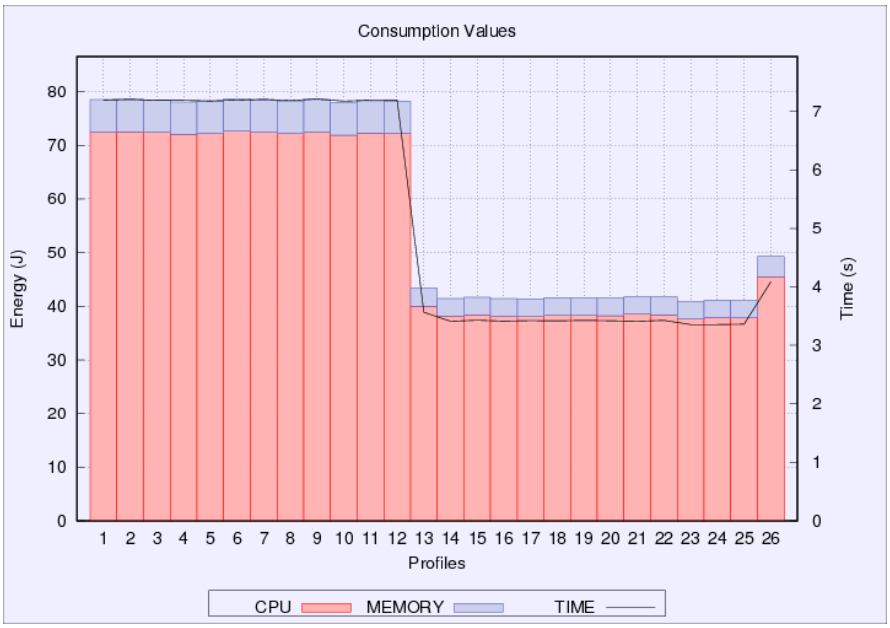


Figure 38.: Results of binary-trees measurements.

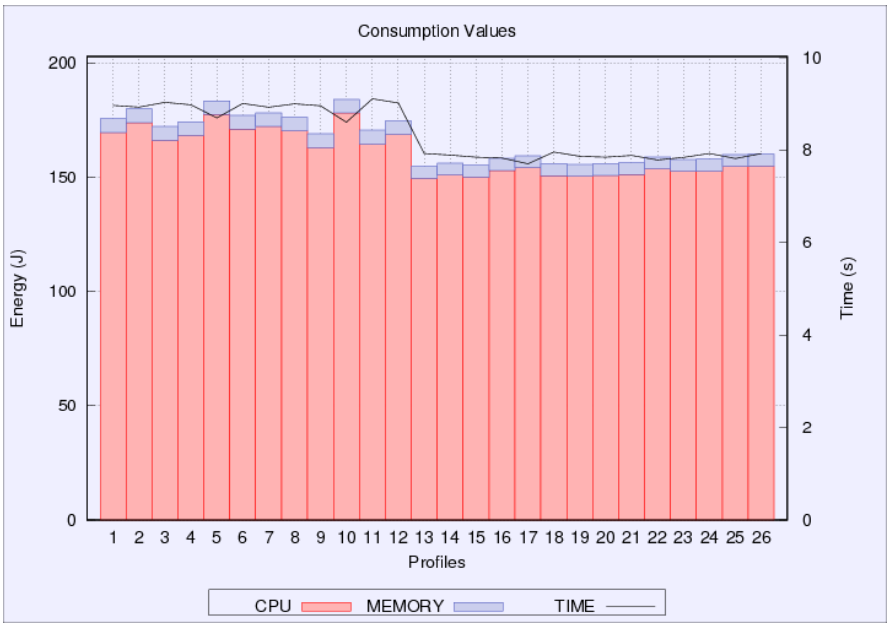


Figure 39.: Results of chameneos-redux measurements.

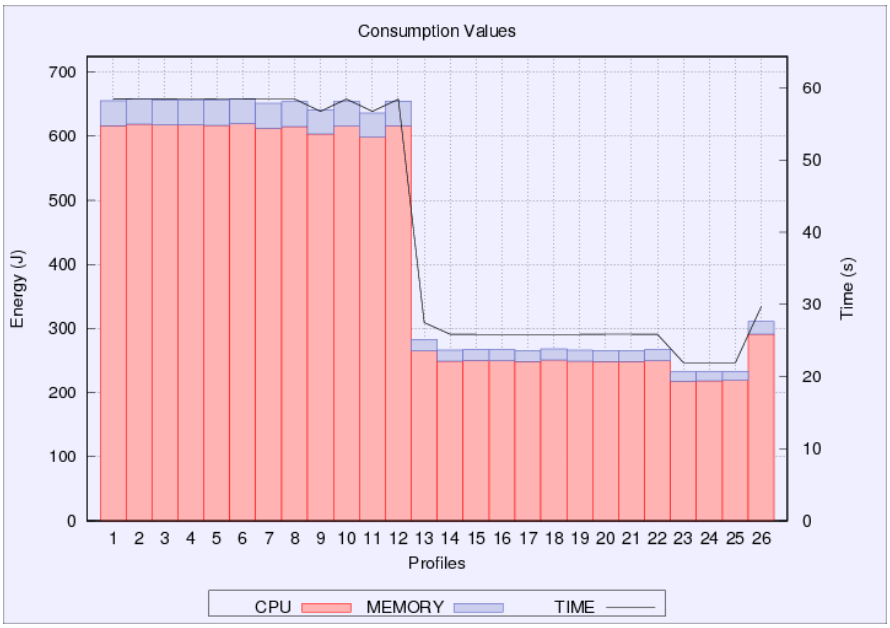


Figure 40.: Results of fannkuch-redux measurements.

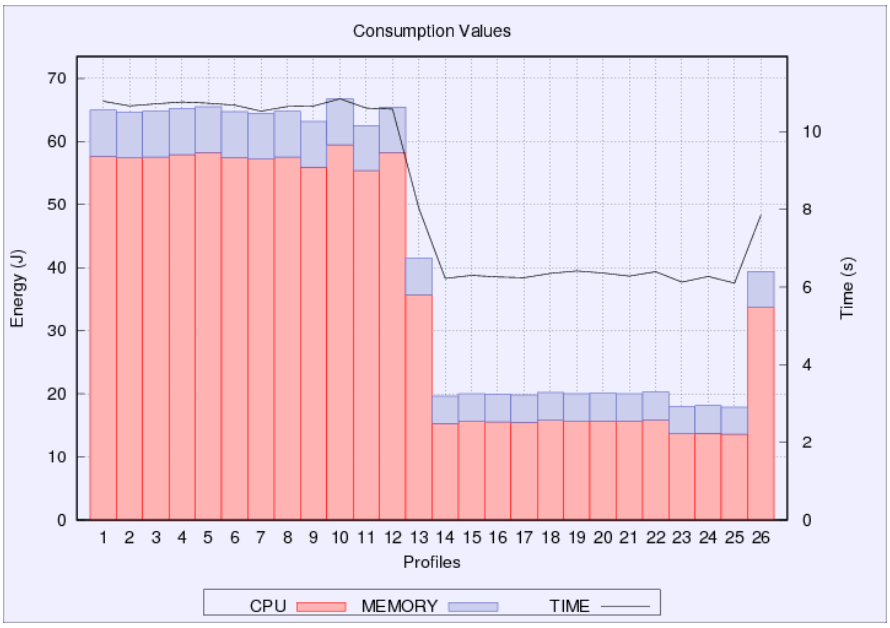


Figure 41.: Results of fasta measurements.

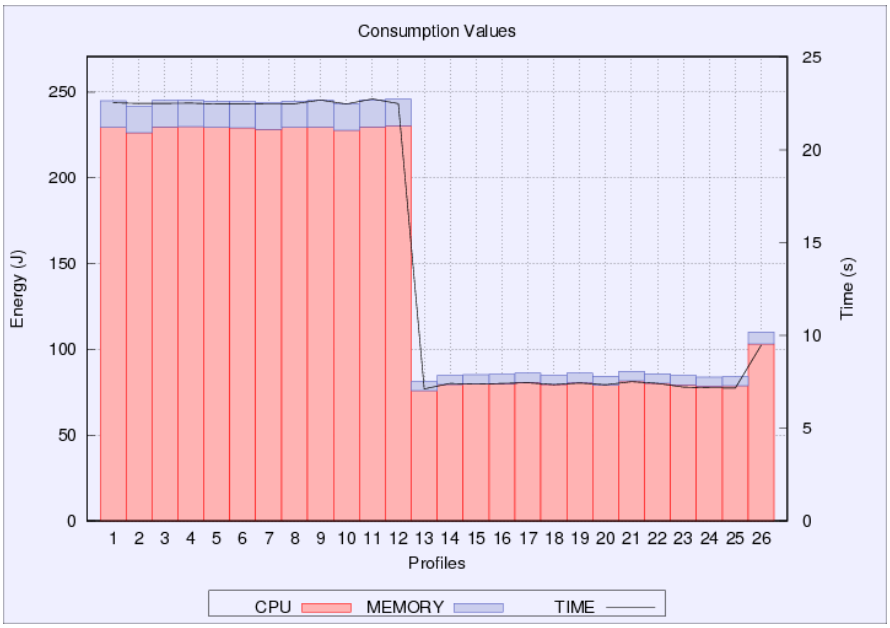


Figure 42.: Results of k-nucleotide measurements.

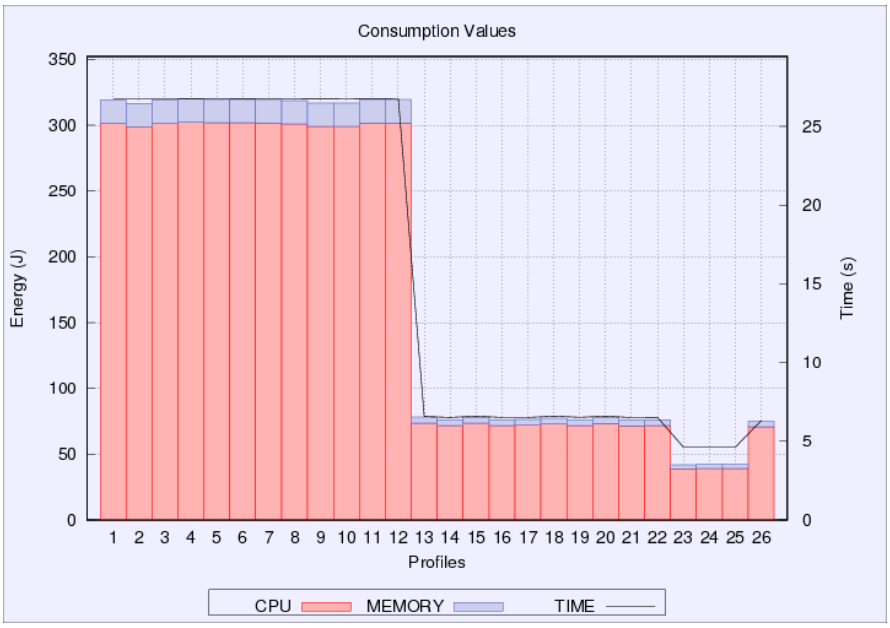


Figure 43.: Results of mandelbrot measurements.

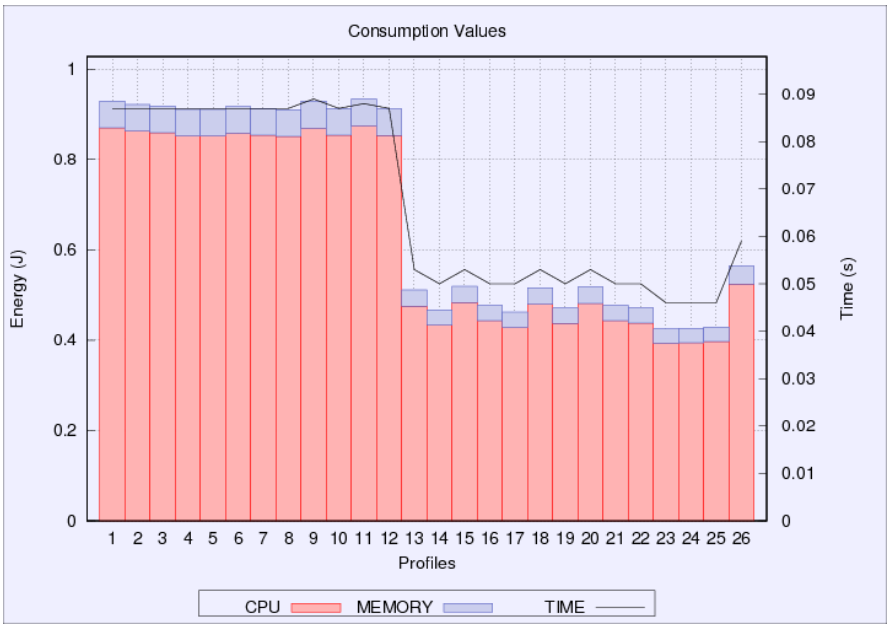


Figure 44.: Results of meteor measurements.

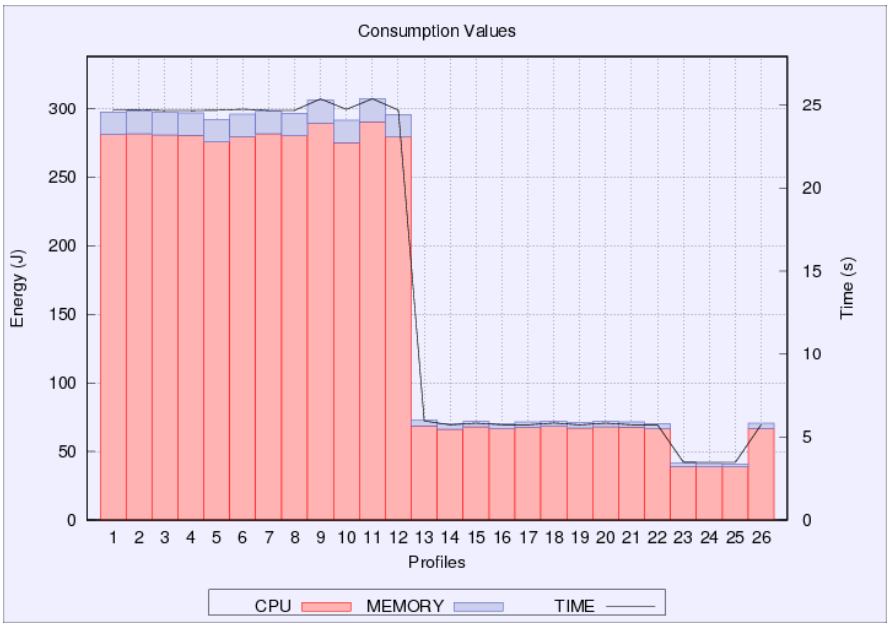


Figure 45.: Results of n-body measurements.

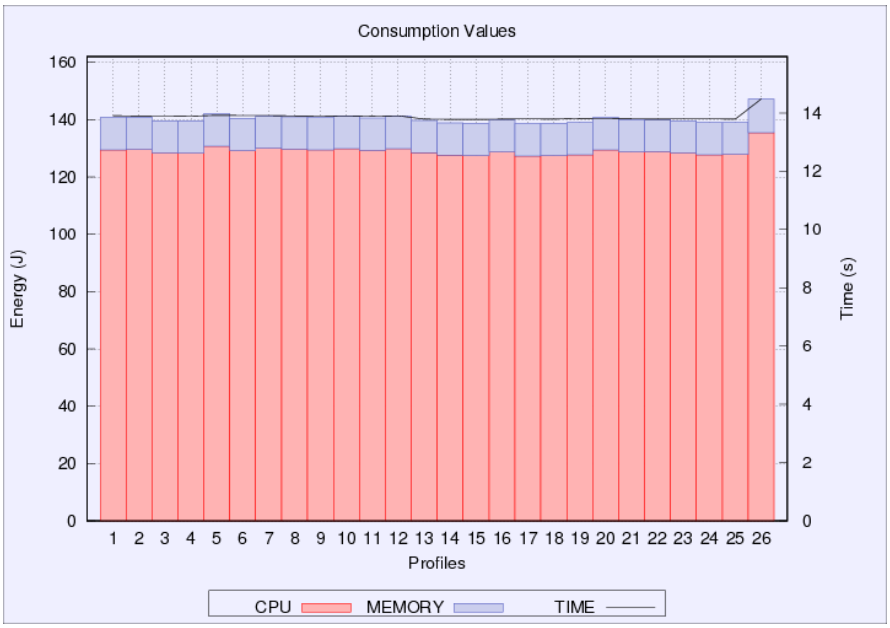


Figure 46.: Results of regex-redux measurements.

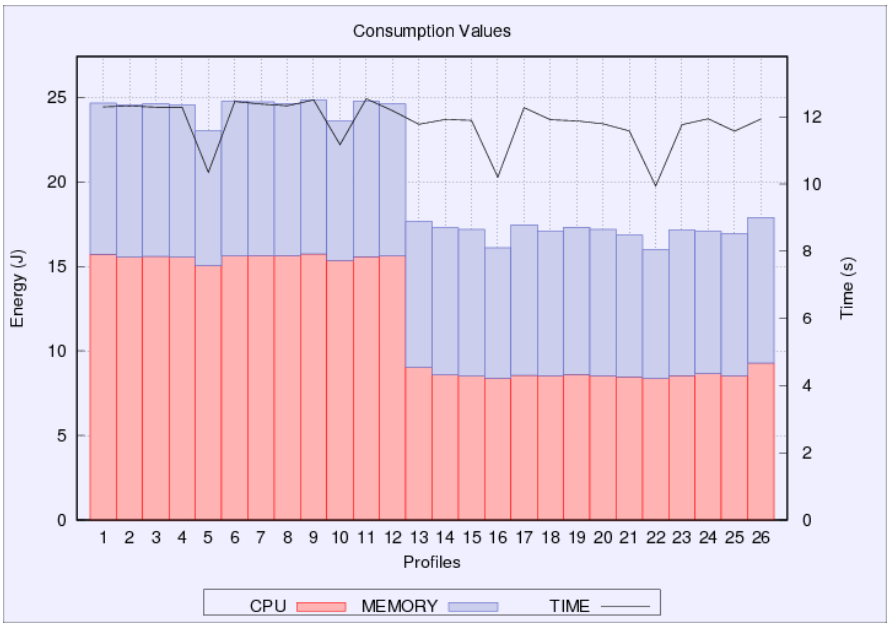


Figure 47.: Results of reverse-complement measurements.

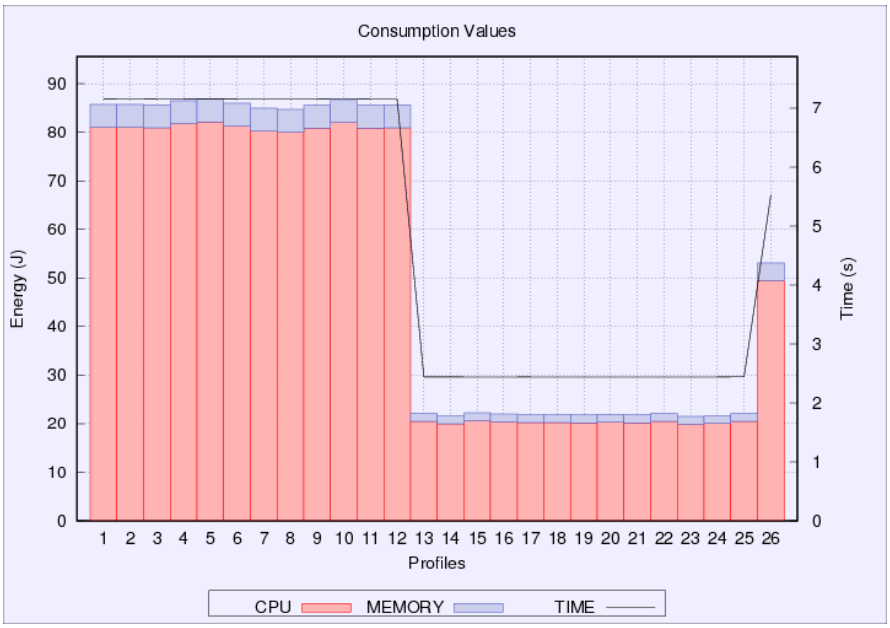


Figure 48.: Results of spectral-norm measurements.

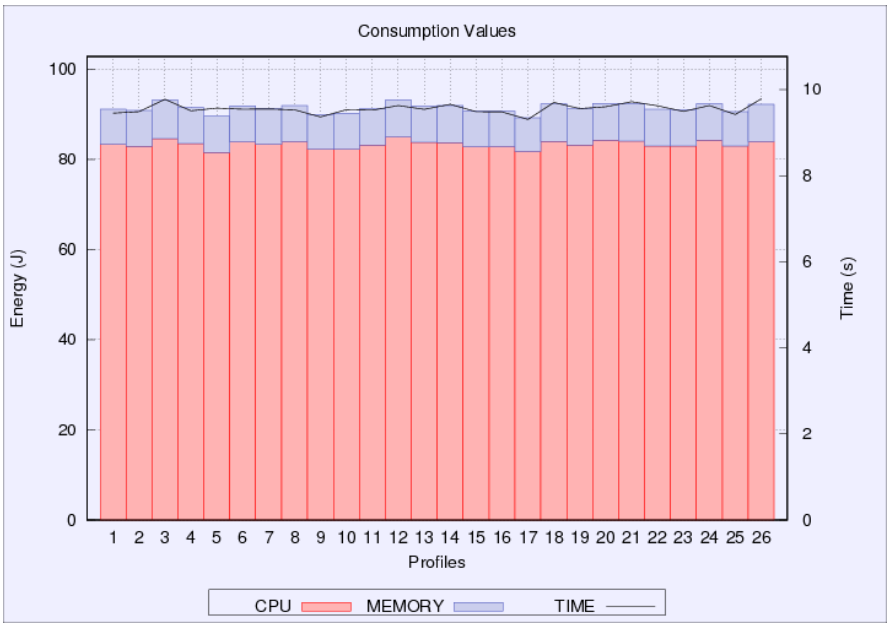


Figure 49.: Results of thread-ring measurements.

Tool Name	Execution Time (s)	Total Energy (J)	CPU Energy (J)	Memory Energy (J)	Energy/Time (J/s)
CMake	3 (23.2)	3 (43.5)	3 (44.5)	3 (21.2)	3 (19.0)
qmake	4 (24.0)	4 (44.0)	4 (45.0)	4 (21.7)	3 (19.0)
Qbs	9 (27.5)	7 (52.0)	7 (53.0)	10 (24.5)	5 (19.5)
NetBeans IDE	8 (27.0)	10 (55.0)	9 (56.0)	9 (24.0)	6 (20.0)
Code::Blocks	12 (35.0)	12 (80.0)	11 (82.0)	13 (33.0)	10 (23.0)
CLion	3 (23.2)	3 (43.5)	3 (44.5)	3 (21.2)	3 (19.0)
CodeLite	6 (25.5)	7 (52.0)	8 (54.5)	8 (23.5)	7 (20.5)
Eclipse CDT	6 (25.5)	6 (49.0)	6 (50.5)	5 (22.0)	5 (19.5)
KDevelop	3 (23.2)	3 (43.5)	3 (44.5)	3 (21.2)	3 (19.0)
Geany	13 (36.0)	13 (81.0)	13 (85.0)	12 (32.0)	10 (23.0)
Anjuta DevStudio	10 (31.5)	11 (66.0)	10 (68.5)	11 (28.5)	8 (20.8)
Qt Creator	5 (24.4)	5 (45.6)	5 (46.6)	6 (22.1)	4 (19.1)
DialogBlocks	7 (26.5)	8 (53.0)	9 (56.0)	9 (24.0)	5 (19.5)
ZinjaI	6 (25.5)	9 (54.0)	8 (54.5)	7 (23.0)	5 (19.5)
GPS	2 (22.5)	2 (41.2)	2 (42.0)	2 (19.8)	2 (18.8)
Oracle Developer Studio	8 (27.0)	10 (55.0)	9 (56.0)	9 (24.0)	6 (20.0)
Sphere Engine	1 (20.0)	1 (30.0)	1 (32.0)	1 (16.0)	1 (18.0)
AWS Cloud9	11 (34.0)	14 (81.5)	12 (83.5)	12 (32.0)	9 (22.5)

Table 16.: Tools ranked with 0 decimal points.

Tool Name	Execution Time (s)	Total Energy (J)	CPU Energy (J)	Memory Energy (J)	Energy/Time (J/s)
CMake	3 (42.2)	3 (101.2)	3 (96.2)	3 (40.8)	3 (46.8)
qmake	8 (46.3)	4 (105.7)	4 (99.3)	6 (43.3)	4 (47.0)
Qbs	11 (51.5)	9 (124.5)	9 (119.5)	11 (50.5)	7 (50.0)
NetBeans IDE	10 (50.5)	11 (127.0)	9 (119.5)	10 (48.5)	7 (50.0)
Code::Blocks	15 (66.0)	15 (179.0)	14 (174.0)	15 (61.0)	11 (59.0)
CLion	3 (42.2)	3 (101.2)	3 (96.2)	3 (40.8)	3 (46.8)
CodeLite	6 (45.0)	7 (118.0)	7 (114.0)	7 (43.5)	8 (50.5)
Eclipse CDT	5 (44.5)	6 (112.5)	6 (108.0)	5 (43.0)	6 (49.0)
KDevelop	3 (42.2)	3 (101.2)	3 (96.2)	3 (40.8)	3 (46.8)
Geany	14 (64.0)	13 (170.0)	12 (165.0)	14 (60.0)	12 (62.0)
Anjuta DevStudio	12 (58.2)	12 (150.5)	11 (143.0)	12 (55.5)	9 (52.0)
Qt Creator	7 (45.7)	5 (107.9)	5 (102.4)	8 (43.8)	5 (47.6)
DialogBlocks	9 (47.0)	10 (125.5)	10 (120.0)	9 (45.5)	9 (52.0)
ZinjaI	4 (43.0)	8 (121.5)	8 (116.5)	4 (41.5)	10 (58.0)
GPS	2 (39.0)	2 (96.2)	2 (91.5)	2 (36.8)	2 (46.5)
Oracle Developer Studio	10 (50.5)	11 (127.0)	9 (119.5)	10 (48.5)	7 (50.0)
Sphere Engine	1 (34.0)	1 (81.0)	1 (79.0)	1 (35.0)	1 (44.0)
AWS Cloud9	13 (62.0)	14 (178.5)	13 (173.0)	13 (56.5)	13 (63.5)

Table 17.: Tools ranked with 1 decimal point.

Tool Name	Execution Time (s)	Total Energy (J)	CPU Energy (J)	Memory Energy (J)	Energy/Time (J/s)
CMake	4 (87.2)	3 (128.8)	3 (127.5)	4 (81.2)	2 (109.8)
qmake	6 (92.7)	4 (136.7)	4 (133.7)	7 (87.0)	3 (112.3)
Qbs	10 (109.5)	11 (164.0)	11 (160.5)	11 (103.0)	7 (119.0)
NetBeans IDE	9 (108.0)	10 (162.0)	10 (158.0)	10 (99.5)	6 (116.0)
Code::Blocks	14 (137.0)	15 (226.0)	15 (231.0)	14 (120.0)	12 (143.0)
CLion	4 (87.2)	3 (128.8)	3 (127.5)	4 (81.2)	2 (109.8)
CodeLite	5 (88.0)	7 (148.5)	7 (151.0)	3 (80.0)	9 (122.0)
Eclipse CDT	8 (96.0)	6 (139.5)	6 (144.0)	6 (86.0)	5 (113.5)
KDevelop	4 (87.2)	3 (128.8)	3 (127.5)	4 (81.2)	2 (109.8)
Geany	13 (133.0)	13 (216.0)	13 (214.0)	15 (121.0)	11 (139.0)
Anjuta DevStudio	11 (123.0)	12 (188.8)	12 (189.0)	12 (111.8)	8 (121.8)
Qt Creator	7 (94.0)	5 (139.2)	5 (136.9)	8 (88.0)	4 (112.7)
DialogBlocks	7 (94.0)	9 (156.0)	9 (156.0)	9 (89.0)	9 (122.0)
ZinjaI	3 (87.0)	8 (153.0)	8 (152.5)	5 (83.0)	10 (126.5)
GPS	2 (76.5)	2 (121.8)	2 (123.2)	2 (71.2)	5 (113.5)
Oracle Developer Studio	9 (108.0)	10 (162.0)	10 (158.0)	10 (99.5)	6 (116.0)
Sphere Engine	1 (71.0)	1 (103.0)	1 (106.0)	1 (67.0)	1 (107.0)
AWS Cloud9	12 (127.0)	14 (223.5)	14 (224.5)	13 (114.0)	13 (148.0)

Table 18.: Tools ranked with 2 decimal points.

Tool Name	Execution Time (s)	Total Energy (J)	CPU Energy (J)	Memory Energy (J)	Energy/Time (J/s)
CMake	3 (123.0)	3 (137.2)	3 (136.5)	3 (119.0)	3 (142.8)
qmake	6 (129.3)	4 (146.3)	4 (144.7)	5 (126.0)	2 (140.7)
Qbs	10 (153.5)	11 (177.5)	11 (173.0)	10 (152.0)	8 (155.5)
NetBeans IDE	10 (153.5)	10 (172.5)	10 (169.0)	9 (148.5)	7 (150.5)
Code::Blocks	13 (200.0)	15 (243.0)	15 (249.0)	14 (193.0)	14 (187.0)
CLion	3 (123.0)	3 (137.2)	3 (136.5)	3 (119.0)	3 (142.8)
CodeLite	4 (123.5)	7 (157.0)	7 (159.5)	4 (123.0)	11 (160.0)
Eclipse CDT	9 (137.5)	5 (148.5)	6 (153.5)	7 (130.0)	5 (145.0)
KDevelop	3 (123.0)	3 (137.2)	3 (136.5)	3 (119.0)	3 (142.8)
Geany	14 (201.0)	13 (232.0)	13 (231.0)	13 (191.0)	13 (181.0)
Anjuta DevStudio	11 (176.8)	12 (201.5)	12 (201.8)	11 (171.5)	9 (156.2)
Qt Creator	7 (131.9)	6 (149.2)	5 (147.3)	6 (128.7)	4 (144.9)
DialogBlocks	8 (136.0)	9 (164.5)	9 (164.5)	8 (136.0)	10 (157.0)
ZinjaI	5 (128.5)	8 (163.5)	8 (162.5)	7 (130.0)	12 (163.0)
GPS	2 (110.0)	2 (130.2)	2 (132.2)	2 (107.0)	6 (147.8)
Oracle Developer Studio	10 (153.5)	10 (172.5)	10 (169.0)	9 (148.5)	7 (150.5)
Sphere Engine	1 (95.0)	1 (110.0)	1 (110.0)	1 (96.0)	1 (137.0)
AWS Cloud9	12 (188.5)	14 (237.5)	14 (240.0)	12 (185.0)	15 (193.0)

Table 19.: Tools ranked with 3 decimal points.

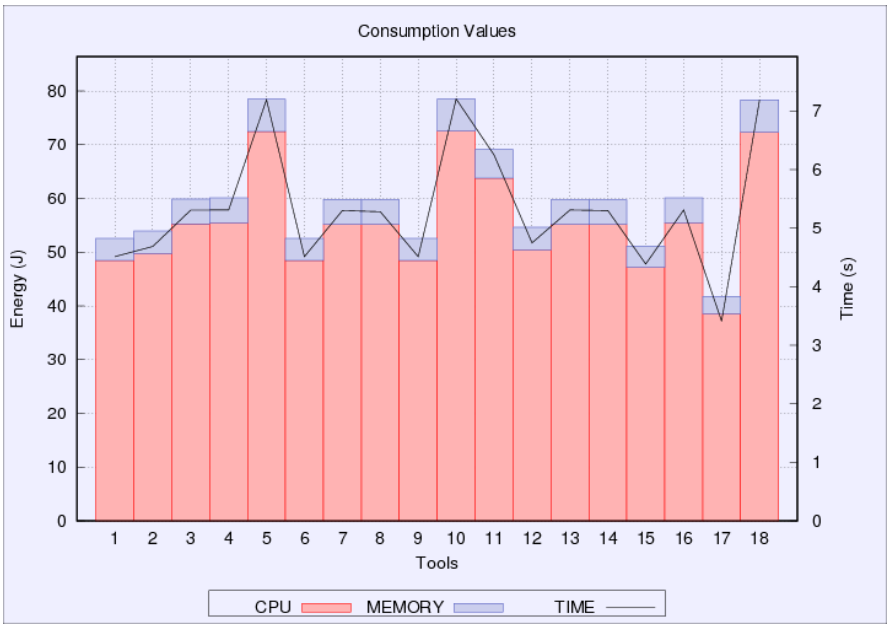


Figure 50.: Tools measurements for binary-trees.

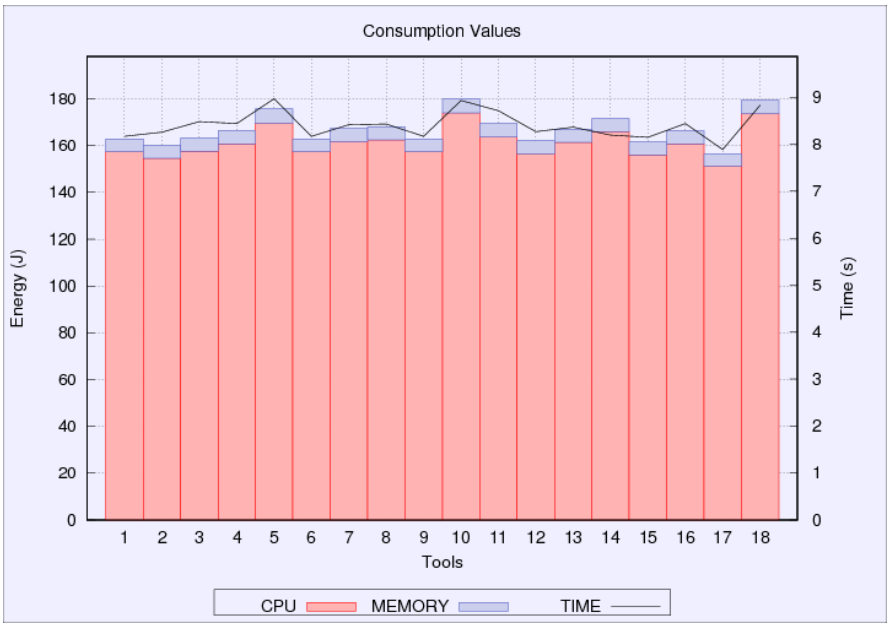


Figure 51.: Tools measurements for chameneos-redux.

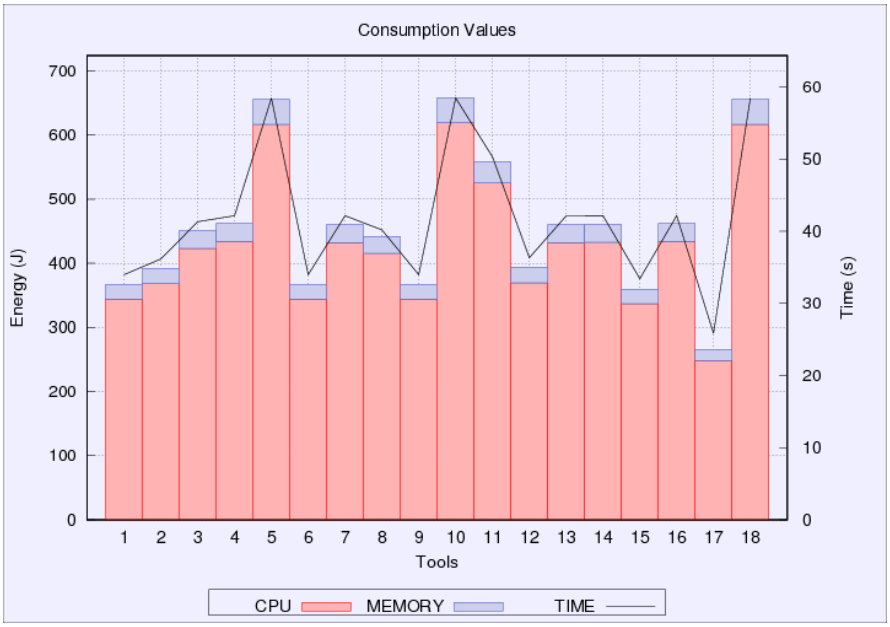


Figure 52.: Tools measurements for fannkuch-redux.

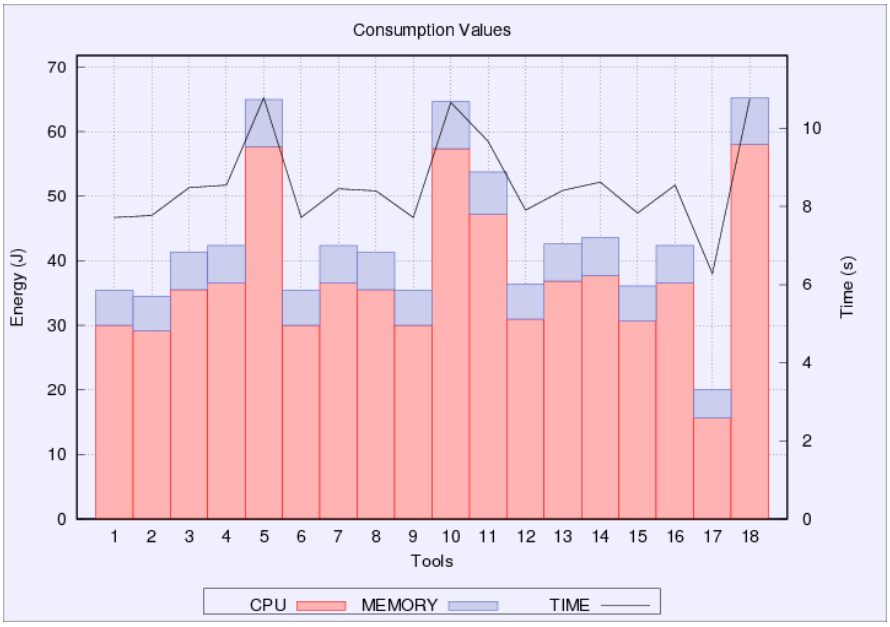


Figure 53.: Tools measurements for fasta.

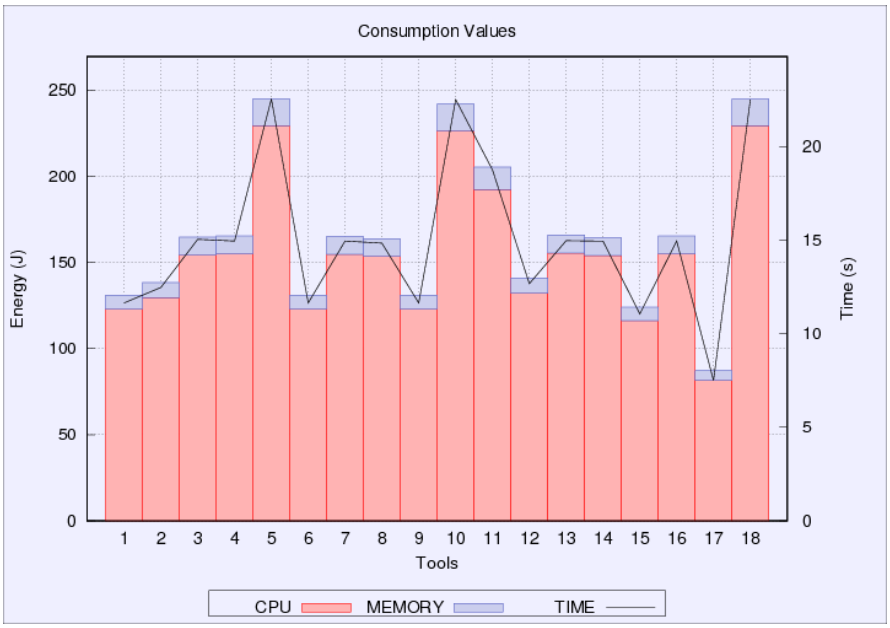


Figure 54.: Tools measurements for k-nucleotide.

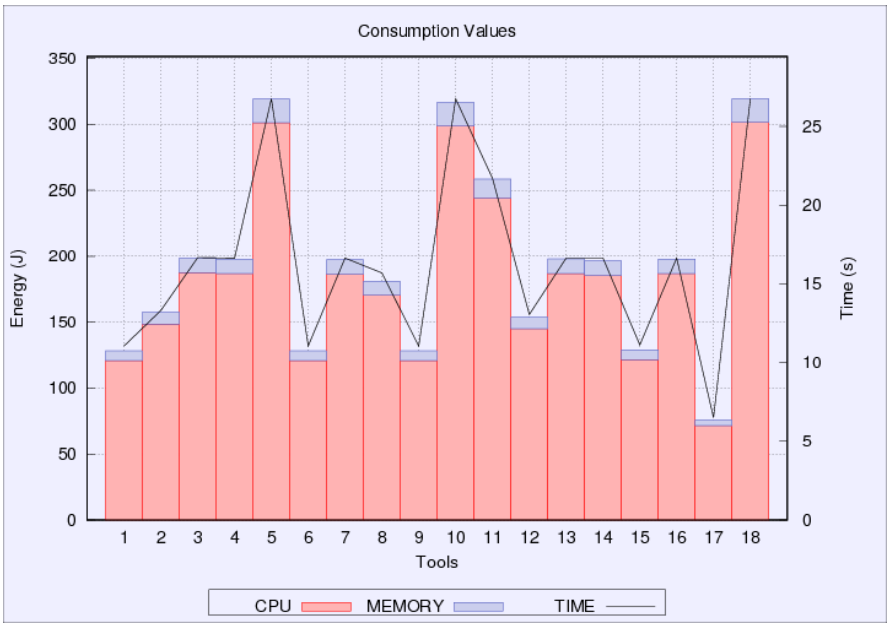


Figure 55.: Tools measurements for mandelbrot.

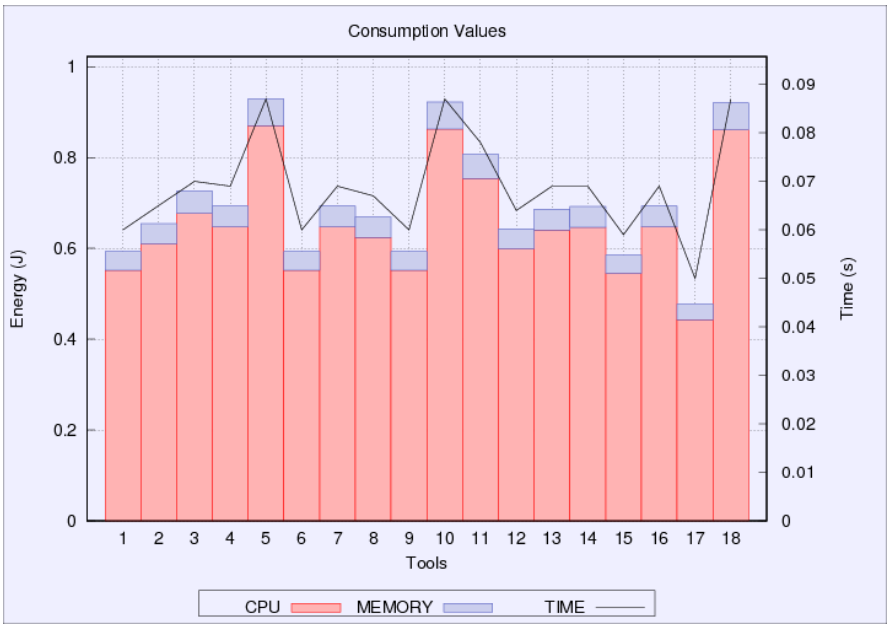


Figure 56.: Tools measurements for meteor.

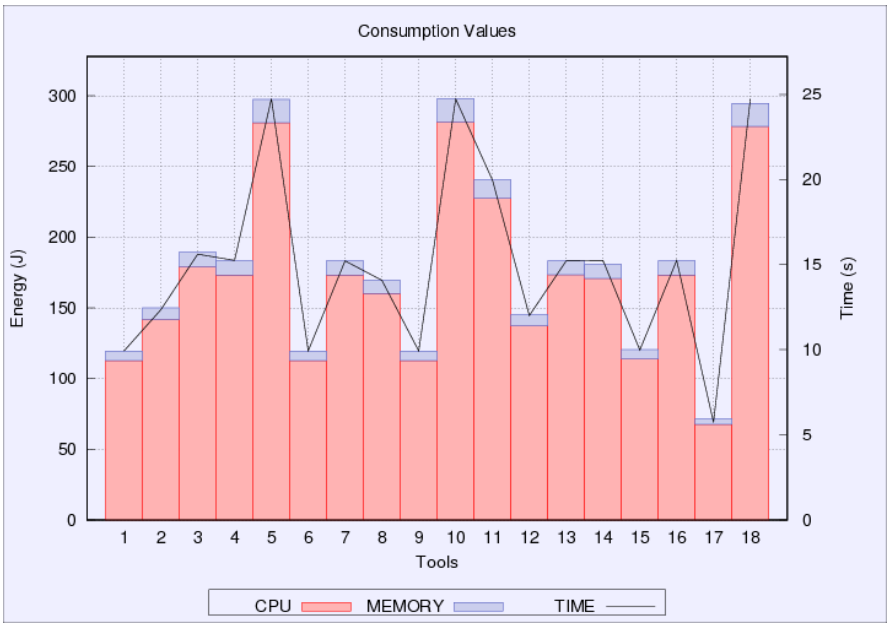


Figure 57.: Tools measurements for n-body.

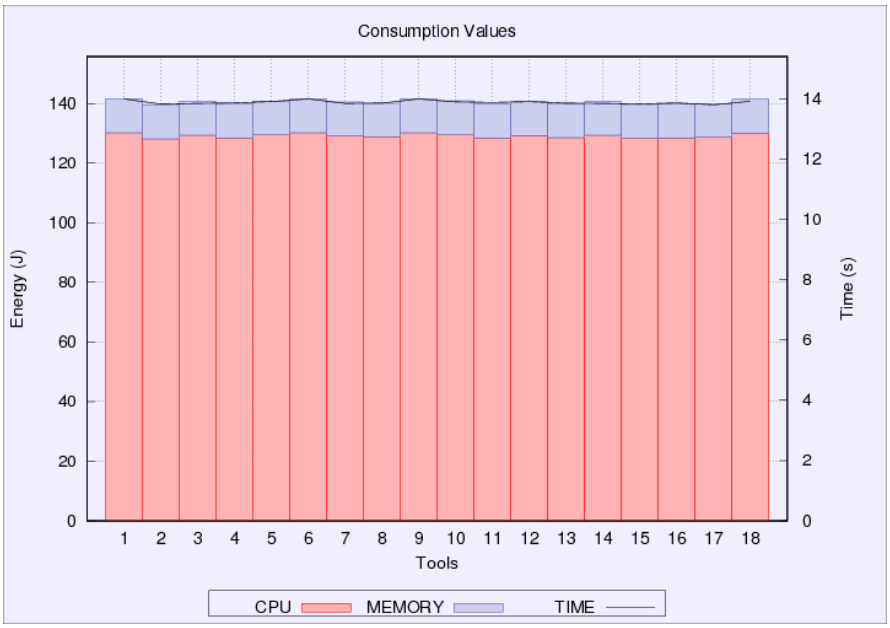


Figure 58.: Tools measurements for regex-redux.

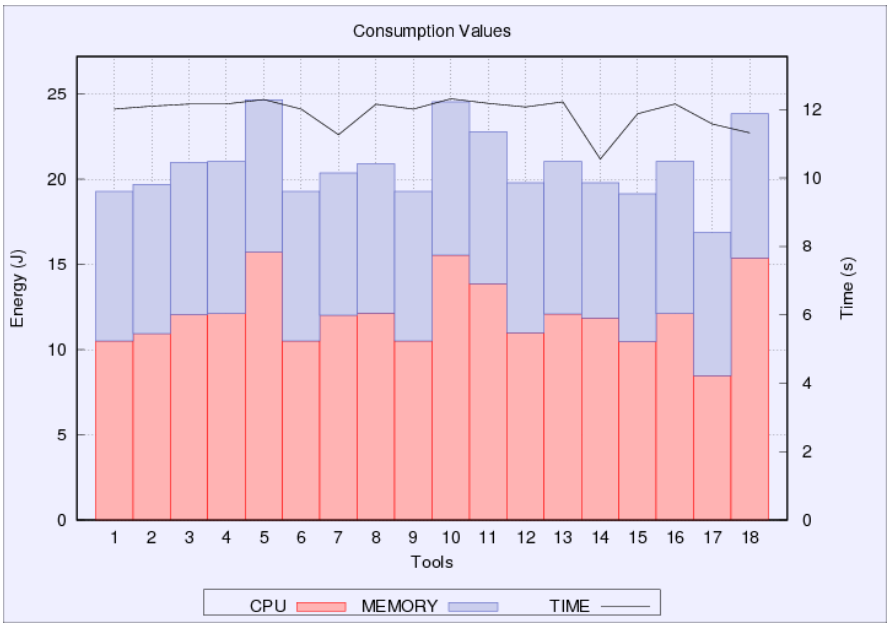


Figure 59.: Tools measurements for reverse-complement.

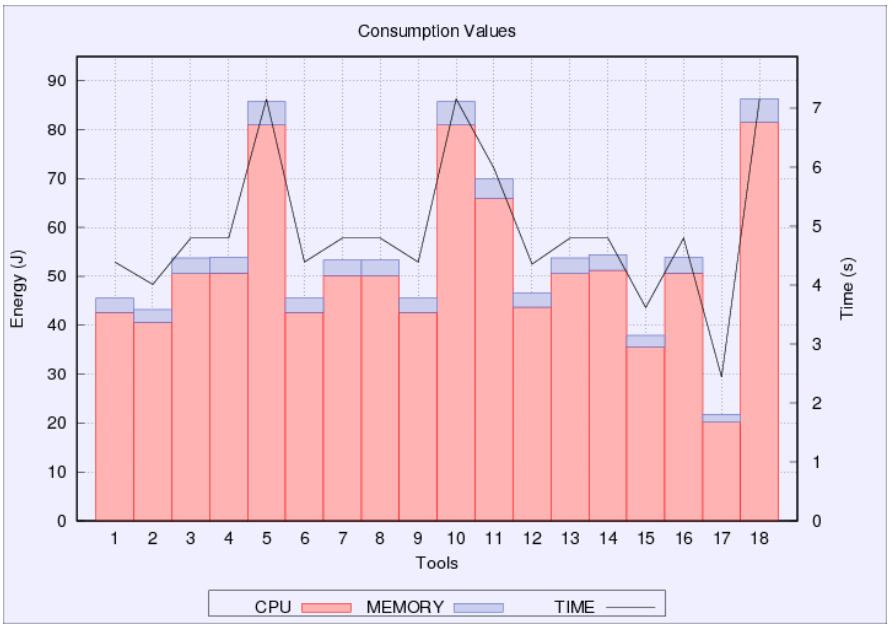


Figure 60.: Tools measurements for spectral-norm.

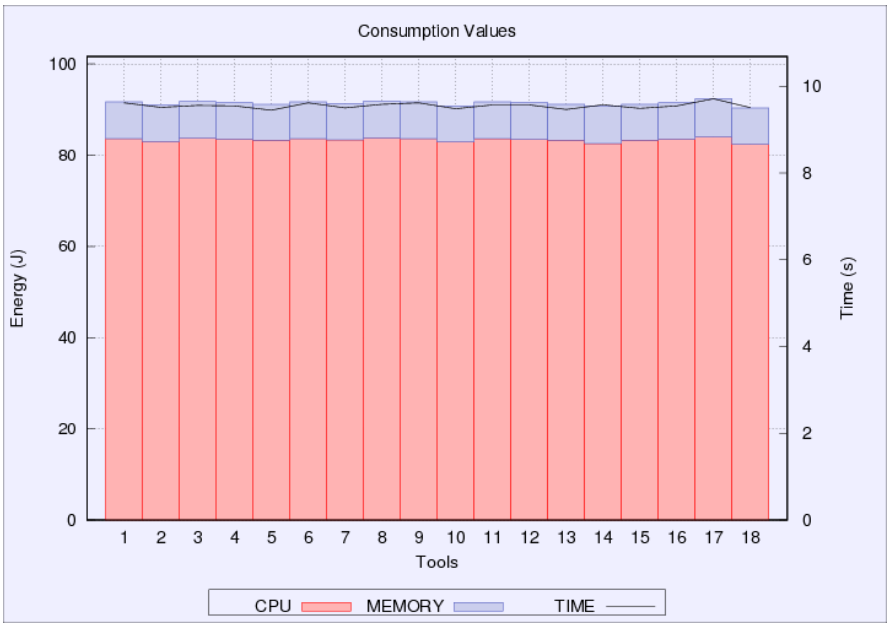


Figure 61.: Tools measurements for thread-ring.

Profile ID	Execution Time (s)	Total Energy (J)	CPU Energy (J)	Memory Energy (J)	Energy/Time (J/s)
1	12 (35)	15 (80)	17 (82)	9 (33)	8 (23)
2	13 (36)	16 (81)	20 (85)	8 (32)	8 (23)
3	13 (36)	13 (78)	16 (81)	9 (33)	7 (22)
4	13 (36)	14 (79)	18 (83)	8 (32)	6 (21)
5	10 (33)	17 (83)	20 (85)	7 (31)	7 (22)
6	12 (35)	17 (83)	19 (84)	8 (32)	8 (23)
7	13 (36)	13 (78)	16 (81)	8 (32)	7 (22)
8	12 (35)	13 (78)	15 (80)	8 (32)	7 (22)
9	12 (35)	11 (70)	12 (72)	9 (33)	6 (21)
10	11 (34)	14 (79)	14 (79)	7 (31)	7 (22)
11	13 (36)	12 (72)	13 (75)	9 (33)	7 (22)
12	12 (35)	15 (80)	18 (83)	8 (32)	6 (21)
13	8 (22)	9 (37)	10 (36)	5 (18)	3 (18)
14	6 (19)	3 (25)	4 (26)	4 (16)	1 (16)
15	5 (18)	7 (30)	8 (31)	4 (16)	3 (18)
16	3 (16)	4 (26)	6 (29)	3 (15)	4 (19)
17	5 (18)	4 (26)	6 (29)	4 (16)	3 (18)
18	6 (19)	8 (32)	9 (32)	4 (16)	3 (18)
19	6 (19)	5 (27)	5 (28)	4 (16)	2 (17)
20	6 (19)	8 (32)	8 (31)	4 (16)	2 (17)
21	7 (20)	7 (30)	9 (32)	4 (16)	3 (18)
22	4 (17)	6 (29)	7 (30)	3 (15)	2 (17)
23	1 (14)	1 (20)	1 (19)	2 (13)	2 (17)
24	2 (15)	1 (20)	2 (20)	1 (12)	2 (17)
25	1 (14)	2 (21)	3 (22)	1 (12)	2 (17)
26	9 (25)	10 (46)	11 (49)	6 (24)	5 (20)

Table 20.: Profiles ranked with 0 decimal points.

Profile ID	Execution Time (s)	Total Energy (J)	CPU Energy (J)	Memory Energy (J)	Energy/Time (J/s)
1	15 (66)	20 (179)	22 (174)	15 (61)	17 (59)
2	13 (64)	17 (170)	18 (165)	14 (60)	19 (62)
3	17 (68)	18 (177)	19 (169)	19 (66)	13 (53)
4	14 (65)	18 (177)	19 (169)	16 (62)	15 (57)
5	10 (58)	19 (178)	21 (172)	12 (52)	20 (68)
6	16 (67)	21 (185)	23 (179)	18 (64)	18 (61)
7	12 (63)	18 (177)	20 (170)	16 (62)	18 (61)
8	13 (64)	16 (168)	17 (162)	16 (62)	14 (56)
9	16 (67)	14 (158)	14 (152)	17 (63)	12 (52)
10	11 (59)	15 (165)	15 (158)	13 (56)	20 (68)
11	18 (69)	16 (168)	16 (160)	19 (66)	13 (53)
12	13 (64)	21 (185)	23 (179)	17 (63)	16 (58)
13	7 (37)	12 (92)	12 (87)	10 (36)	6 (43)
14	6 (34)	4 (63)	4 (56)	7 (32)	1 (35)
15	6 (34)	8 (76)	8 (69)	7 (32)	10 (47)
16	3 (26)	6 (68)	7 (66)	4 (25)	8 (45)
17	5 (30)	5 (66)	6 (61)	6 (28)	9 (46)
18	8 (38)	11 (83)	10 (77)	9 (35)	5 (42)
19	6 (34)	7 (69)	5 (60)	8 (33)	4 (39)
20	6 (34)	10 (81)	11 (79)	9 (35)	10 (47)
21	6 (34)	10 (81)	11 (79)	9 (35)	7 (44)
22	4 (27)	9 (78)	9 (75)	5 (27)	11 (48)
23	2 (21)	1 (38)	1 (37)	2 (22)	3 (38)
24	3 (26)	3 (48)	3 (46)	3 (24)	2 (37)
25	1 (19)	2 (45)	2 (45)	1 (17)	8 (45)
26	9 (47)	13 (119)	13 (113)	11 (45)	13 (53)

Table 21.: Profiles ranked with 1 decimal point.

Profile ID	Execution Time (s)	Total Energy (J)	CPU Energy (J)	Memory Energy (J)	Energy/Time (J/s)
1	19 (137)	23 (226)	22 (231)	18 (120)	18 (143)
2	16 (133)	18 (216)	16 (214)	19 (121)	16 (139)
3	22 (149)	22 (224)	21 (224)	25 (139)	11 (123)
4	16 (133)	19 (218)	19 (219)	16 (118)	14 (132)
5	14 (117)	20 (221)	18 (218)	14 (108)	19 (153)
6	20 (143)	25 (237)	23 (234)	22 (129)	18 (143)
7	16 (133)	21 (222)	20 (223)	20 (122)	17 (140)
8	16 (133)	16 (211)	17 (215)	17 (119)	13 (130)
9	17 (134)	14 (204)	14 (202)	21 (124)	10 (119)
10	15 (118)	15 (205)	14 (202)	15 (112)	15 (137)
11	21 (147)	17 (213)	15 (207)	24 (135)	11 (123)
12	18 (135)	24 (235)	24 (237)	23 (130)	18 (143)
13	9 (71)	12 (123)	12 (123)	10 (70)	8 (115)
14	8 (67)	4 (77)	4 (73)	8 (63)	1 (81)
15	7 (62)	7 (98)	8 (93)	7 (61)	9 (116)
16	3 (43)	5 (86)	7 (87)	3 (41)	7 (114)
17	4 (53)	4 (77)	5 (75)	4 (48)	4 (101)
18	12 (82)	10 (108)	10 (106)	12 (76)	5 (102)
19	11 (73)	6 (87)	6 (82)	10 (70)	3 (89)
20	10 (72)	11 (115)	11 (114)	11 (71)	8 (115)
21	9 (71)	9 (103)	10 (106)	9 (67)	6 (107)
22	5 (56)	8 (101)	9 (103)	6 (54)	9 (116)
23	2 (36)	1 (45)	1 (48)	2 (38)	2 (87)
24	6 (59)	3 (57)	3 (65)	5 (50)	2 (87)
25	1 (25)	2 (51)	2 (57)	1 (25)	6 (107)
26	13 (103)	13 (156)	13 (155)	13 (95)	12 (128)

Table 22.: Profiles ranked with 2 decimal points.

Profile ID	Execution Time (s)	Total Energy (J)	CPU Energy (J)	Memory Energy (J)	Energy/Time (J/s)
1	20 (200)	23 (243)	21 (249)	19 (193)	19 (187)
2	21 (201)	20 (232)	18 (231)	18 (191)	18 (181)
3	25 (217)	22 (240)	20 (239)	22 (210)	13 (156)
4	17 (190)	18 (230)	18 (231)	16 (185)	15 (170)
5	15 (177)	20 (232)	18 (231)	14 (177)	22 (199)
6	24 (207)	25 (252)	22 (250)	21 (199)	21 (189)
7	23 (204)	21 (237)	19 (238)	20 (198)	17 (179)
8	18 (192)	17 (222)	17 (227)	17 (190)	15 (170)
9	18 (192)	16 (220)	15 (220)	18 (191)	11 (154)
10	16 (179)	15 (218)	14 (217)	15 (184)	16 (172)
11	22 (202)	19 (231)	16 (225)	21 (199)	13 (156)
12	19 (198)	24 (248)	22 (250)	21 (199)	20 (188)
13	12 (107)	13 (130)	12 (131)	11 (104)	10 (153)
14	8 (91)	5 (82)	4 (79)	8 (84)	1 (102)
15	7 (84)	8 (106)	7 (101)	7 (82)	8 (143)
16	3 (55)	6 (92)	6 (92)	3 (56)	9 (150)
17	5 (74)	4 (81)	4 (79)	5 (73)	5 (126)
18	13 (112)	11 (113)	10 (113)	12 (105)	4 (125)
19	10 (100)	7 (93)	5 (88)	10 (98)	3 (112)
20	11 (105)	12 (124)	11 (121)	12 (105)	12 (155)
21	9 (95)	10 (110)	9 (110)	9 (96)	6 (137)
22	6 (78)	9 (109)	8 (108)	6 (76)	11 (154)
23	2 (47)	1 (48)	1 (51)	2 (50)	3 (112)
24	4 (71)	3 (60)	3 (69)	4 (62)	2 (111)
25	1 (33)	2 (55)	2 (61)	1 (33)	7 (139)
26	14 (147)	14 (167)	13 (166)	13 (143)	14 (168)

Table 23.: Profiles ranked with 3 decimal points.