Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages and Compilers

Green computing in Erlang

Supervisor:
Melinda Tóth
Assistant lecturer

Author:
Jessica Tatiana Carrasco Ortiz
Computer Science MSc
2. year

Budapest, 2017

Abstract

Green computing is the environmentally responsible and eco-friendly use of computers and their resources. Erlang is functional programming languages but unfortunately, there is not previous research about Green computing in Erlang. The thesis is proposing a research about it, analyzing the energy behavior in Erlang functions. To achieve that goal has been necessary to create a new software that is available to measure the energy consumption in Erlang. The functions must be refactored in different algorithms in order to show that, depending on either the algorithm or refactoring the energy consumption varies. The results show that Higher order Functions increase the energy consumption, reduce the structure Data in some test reduce the energy consumption and the Introduction of Variables does not affect the energy consumption. However, the energy consumption depends on many factors of the computer and of the environment of the software. Therefore, when the target is reduced the energy consumption it is important to consider all the approach in order to get a general overview.

Contents

Chapter 1

Introduction

In 2016 the first six months set a record as the warmest respective month globally in the modern temperature record, which dates since 1880, also the warmest half-year on record of the planet, with an average temperature 1.3 degrees Celsius warmer than the late nineteenth century [1]. Between 1900 and 2015 the floods have caused around a third of economic losses, the earthquakes have caused around 26 percent of losses, storms around 19 percent, volcanic eruptions around 1 percent [2].

Scientific and Environmental non-governmental organization (ENGO) started to analyze which are the facts that affect our planet. One of the bigger involved is Computing. Due to fact that a simple computer consumes energy, the terminology *Green Computing* appear.

Erlang is a functional language program that in the last years has become popular, big companies are using this language to their advantages. Green Computing in Erlang is a new topic, there are not previous research about it. Nevertheless, recent work has studied the effect of energy consumption in other functional language programs like Haskell and other has studied mechanisms how to reduce the energy consumption in languages like Java, C++, C, etc. Analyzing those research were gotten ideas to measure the energy consumption in Erlang and the types of refactoring to applied.

To measure the energy that Erlang functions produce is necessary the creation of a new software. The functions must be refactored in different algorithms to measure and, show that with the same results the energy consumption varies depending which algorithm is used. It is possible to get some patterns or recommendations based on those tests, they will help to developers to create functions that consume less energy. Obviously, the energy consumption depends on many facts on the computer, so this pattern or recommendations has been analyzed with the environment but not considered rules.

Chapter 2

Background

Next the principal concepts on which the thesis is based.

2.1 Green Computing

Green computing is the environmentally responsible and Eco-friendly use of computers and their resources. Defined as the study of designing, manufacturing, engineering and re-engineering, using and disposing of computing devices in a way that reduces their environmental impact [3]. It implies the implementation of energy-efficient in the central processing units (CPUs), servers and peripherals in order to reduce resource consumption and proper disposal of electronic waste[4].

Green computing practices came into prominence in 1992, when the Environmental Protection Agency (EPA) launched the Energy Star program. Actually, many IT manufacturers and vendors are investing in designing energy efficient computing devices, reducing the use of dangerous materials and encouraging the recyclability of digital devices and paper [3].

Think about the colossal amount of computing manufactured worldwide has a direct impact on environmental issues, for this reason, the scientists are making numerous studies in order to reduce the negative impact of computing technology on our natural resources. Around the world, big companies started to think about green computing and now they are addressing e-waste by offering take-back recycling programs and other solutions, with lower energy consumption and less wasted hardware. The principal idea is tested and applying alternative materials in the products manufacturing process [4].

Green computing is working in software and hardware levels. Regards to hardware has a wide of fields where actually is working, for example [5]:

- Using energy-efficient hardware like notebook computers, displays, servers, printers and desktop computers.

- Better use of resources, such as reduced paper consumption and lower energy utilization.
- Greater awareness so that technology components do not end up in the waste stream.
- Stronger environmental controls for technology production, leading to fewer toxic chemicals in the finished products.

Is important highlight that the biggest companies like Microsoft, Google are working on that. Microsoft, which consumes up to 27 megawatts of energy at any given time has built data centers in central Washington to take advantage of the hydroelectric power produced by two dams in the region. Another Microsoft data center, located in Dublin, Ireland, thanks to moderate climate in Ireland, the 51,000-square-meter facility will be air cooled, making it 50% more efficient than other comparably sized data centers.

Another big Google has committed to being carbon-neutral for 2007 and beyond. The carbon footprint is calculated globally and includes our direct fuel use, purchased electricity, and business travel—as well as estimates for employee commuting, construction, and server manufacturing at our facilities around the world. According to Google, its data centers use half the average amount of power of industry. Google attributes this improved energy usage of the cooling technologies, such as ultra-efficient evaporative cooling, that the company has customized for itself [5].

Meanwhile all forms of hardware systems are controlled by software components. Therefore green computing is used to denote the efficient use of resources in computing in software level,too. This term generally relates to the use of computing resources in conjunction with minimizing environmental impact, maximizing the economic viability and ensuring social duties. Software drives the hardware thus decisions taken during software design and development have significant impact on energy consumption of a computing system. Although software systems do not consume energy directly, they affect hardware utilization, leading to indirect energy consumption.

Green computing in software level play a proactive role in saving energy by providing feedback about the way they consume resources and, ideally, leading people to change behaviors and create greener processes. The software engineering research domain has been paying attention to sustainability, as the increased number of publications, empirical studies, and conferences on the topic demonstrate [6].

2.2 Erlang

Erlang [7] is a programming language used to build massively scalable soft real-time systems with requirements for high availability. Telecoms, banking, e-commerce, computer telephony and instant messaging are using Erlang. Runtime system of Erlang has built-in support for concurrency, distribution and fault tolerance.

Erlang has a set of libraries and design principles providing middleware to develop these systems this is called Open Telecom Platform (OTP). OTP is aimed at providing time-saving and flexible development for robust, adaptable telecom systems. It consists of an Erlang runtime system, a number of ready-to-use components, mainly written in Erlang, and a set of design principles for Erlang programs. Also, it includes its own distributed database, applications to interface

towards other languages, debugging and release handling tools. Since Erlang and OTP are closely interconnected the term Erlang/OTP is normally used instead of OTP [8].

The Erlang Runtime System (ERTS) is made up of an emulator running on top of the host operating system, a kernel providing low-level services such as distribution and I/O handling, and a standard library containing a large number of re-usable modules.

OTP provides the user with a way to structure the system based on a concept called application. An OTP application is a way to package a system component and is either a set of library modules or a supervision tree. A supervision tree is a hierarchical tree of processes used to program fault-tolerant systems. The processes are easiest implemented using behavior modules which are formalizations of design patterns. The standard library includes behavior modules for supervisors, servers, state machines and generic event handlers [9].

The Erlang has the following characteristics [10]:

- Distributed
- Fault-tolerant
- Soft real-time,
- Highly available, non-stop applications
- Hot swapping, where code can be changed without stopping a system.
- The Erlang programming language is known for the following properties:
  - Immutable data
  - Pattern matching
  - Functional programming

2.3 Functional Programming

Functional programming is a programming paradigm that the style of building the structure and elements of computer programs treats the computation as the evaluation of mathematical functions and avoids changing state and mutable data. The programming is done with expressions or declarations instead of statements, it is called declarative programming paradigm. The output value of a function depends only on the arguments that are passed to the function. Functional programming is based on the lambda calculus and provides functions as computational entities [11].

The style of Functional programming is high level thinking about the specification and desired properties of a system, rather than low-level sequential programming of actions to be performed [12].

In contrast, imperative programming has functions in the sense of subroutines, but not in the mathematical sense. They can have side effects that may change the value of program state.

It is mean functions without return values, therefore, make sense because they lack referential transparency, for example the same language expression can result in different values at different times depending on the state of the executing program.

Functional programming languages started to been emphasized in academia rather than in commercial software development. However, prominent programming languages which support functional programmings such as Common Lisp, Scheme, Clojure, Wolfram Language, Racket, Erlang, OCaml, Haskell, and F# have been used in industrial and commercial applications by a wide variety of organizations. Widespread domain-specific declarative languages like SQL and Lex/Yacc use some elements of functional programming, especially in avoiding mutable values.

The increasing importance of parallelism and distribution in computing have made that functional programming has emerged as an attractive basis for software construction since the mid-2000s. In the world of software development where updates require logs or replication for consistency, it is often more efficient to use replayable repayable operations instead of immutable data [13].

Summarizing functional programming is a specific way to look at problems and model their solutions. It is a coding style that exhibits the following characteristics [14]:

- **Power and flexibility.-** We can solve many general real-world problems using functional constructs.
- **Simplicity.-** Most functional programs exhibit a small set of keywords and concise syntax for expressing concepts.
- **Suitable for parallel processing.-** Via immutable values and operators, functional programs lend themselves to asynchronous and parallel processing.

In functional programming, programs are executed by evaluating expressions, in contrast with imperative programming where programs are composed of statements which change global state when executed and typically avoids using mutable state rather everything is a mathematical function. Functional programming languages can have objects, but generally, those objects are immutable either arguments or return values to functions. The type of looping is performed with recursion and by passing functions as arguments.

2.4 Refactoring

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written [15]. Each transformation, even small, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it is less likely to go wrong. The system is kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring [16]. One

of the targets is clean up code that minimizes the chances of introducing bugs. In essence, when you refactor you are improving the design of the code after it has been written.

The comparison of the current behavior of software development and refactoring is that the current behavior design and then code. Therefore a good design comes first, and the coding comes second. Over time the code will be modified, and the integrity of the system, its structure according to that design, gradually fades. But refactoring is the opposite of this practice. Refactoring even can apply in a really bad design, chaos even, and then rework it into well-designed code. Step by step and in a simple way, even simplistic. Refactoring move a field from one class to another, pull some code out of a method, making into its own method, and push some code up or down. All the small changes together can radically improve the design. It is the exact reverse of the current behavior of software development.

Finally refactoring make balance the work changes. The design occurs continuously during development. It improves the design from building the system how. The resulting interaction leads to a program with a design that stays good as development continues [15].

2.5 RefactorErl

RefactorErl is an opensource static source code analyzer and transformer tool for Erlang, developed by the Department of Programming Languages and Compilers at the Faculty of Informatics, Eötvös Loránd University [17].

The main focus of the project is to support daily code comprehension tasks of Erlang developers. While this source code analyzed and transformer tool is still considered as a prototype, its usefulness in industrial usage has already been proved, and every major limitation of real-world usage has been addressed.

The result of different static semantic analysis is available through a user-level semantic query language, that can assist Erlang developers in everyday tasks such as program comprehension, debugging, finding relationships among program parts, etc.

The following are main features of RefactorErl:

- Shorten time-consuming daily jobs.
- Make the possibility of better teamwork in different ways.
- Reduce human faults.
- Easy deploying releases.
- Minimize the training time of newbies.

Chapter 3

Available technologies to measure energy consumption

There are mechanisms either physical and logical to measure the amount of energy that the computer consume. Different softwares have been created in order to measure the energy consumption. One of the interface founded is RAPL(Running Average Power Limit) [18]. RAPL is an interface that is implemented by default in the Linux distributions. Many software are based on RAPL interface, following the principal characteristics of some of them.

3.1 Turbostat

Turbostat is part of the Linux kernel tools and display the wattage information, is reading using RAPL MSRs. Reports processor topology, frequency, temperature, idle power-state statistics and power on x86 processors. There are two ways to invoke turbostat. The first method is to supply a command which is forked and statistics are printed upon its completion. The second methods are to omit the command and turbostat displays statistics every 5-second interval [19].

3.2 PowerTop

PowerTOP is an application developed by Intel that is all about finding the best power settings on Linux systems. PowerTOP, works with all processors as it looks at general settings of not only the processor but other components in your system. Provides estimated power consumption for various components. These estimates are based on real power measurements, taken using a power meter, and a powerful model that takes into account estimated activity on that component. RAPL can help get the exact power numbers for CPUs/GPUs and DRAMs to show actual power consumption. Recent changes in PowerTOP show the actual CPU, GPU, and DRAM power consumption [18].

3.3 Papi

PAPI provides the tool designer and application engineer with a consistent interface and methodology for use of the performance counter hardware found in most major microprocessors. PAPI enables software engineers to see, in near real time, the relation between software performance and processor events. In addition, PAPI provides access to a collection of components that expose performance measurement opportunities across the hardware and software stack [20]. PAPI is a library that gives access to the performance counters in most modern processors. The PAPI library offers a graphical interface where show the energy consumption. To PAPI library need the processor Ivy Bridge [21].

3.4 Perf

Perf is a profiler tool for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple command line interface. Perf is based on the perf_events interface exported by recent versions of the Linux kernel [22].

3.5 Linux Thermal Daemon

RAPL power limits are very effective in reducing package temperature. The Linux thermal daemon uses RAPL interface using RAPL driver to control platform thermals [18]. Proactively Thermal Daemon tries to limit the temperature so BIOS does not take a drastic action to cool the system. The hardware and BIOS can (regulate) itself, but it is usually not an optimal solution, depending on their implementation. And, if the temperature reaches critical point, it can Thermal Daemon diagram results in powering down of the system [23].

3.6 Read_Rapl

RAPL also export the power meters and power limits through a set of MSRs (Machine Specific Registers). Read_RAPL is implemented in C and read the files where is store the amount of energy consumption of the computer. It read these files before a determined period of time and after of this period, in such way that is possible measure the energy consumption of the applications during this period of time. The units of measure that the Read_RAPL use is Joules.

Chapter 4

Measuring Erlang functions

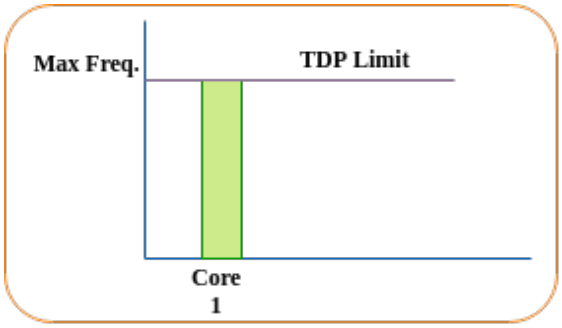
The previous chapter analyzed different software based on RAPL. According to the requirements Read_Rapl is the best option due to the software measure the energy consumption of the computer, in a period of time. The modification of the software and join with Erlang is the software that will measure the Erlang functions. This chapter contains the reason of the creation of RAPL, how it works and how to this thesis the Read_Rapl has been modified to measure the Erlang functions.

It is important remark that at the time that this thesis was made, there is not any tool that could measure the energy consumption in Erlang. In the conferences Erlang refactory a lecture was presented [24], that Filipe Varjão, Ph.D. student of Computer Science at UFPE - Brazil is working on "Measuring Erlang energy consumption", so he is creating a Software that will measure the amount of energy consumption in Erlang. But the software is not yet deployed.

4.1 Why Intel created in RAPL

The thermal design power (TDP) is the maximum amount of power the cooling system in a computer is required to disperse. Hence, Intel guarantees if the computer is using the Original Equipment Manufacturer (OEM) it implements a chassis and cooling system capable of dissipating that much heat, in that way the chip will operate as intended. But the amount of power the cooling system under which the system needs to operate is not the same as the maximum power the processor can consume. The processor could consume more power than the TDP power for a short period in that way that is not really significant "thermally", in such way that the amount of TDP is not affected and is imperceptible. Logically the heat needs some time to propagate, but when the processor used more energy in a short time not necessarily violates TDP.

Figure 4.1: Visualization of TDP in a single Core CPU



As shown the Figure 4.1 In the case that is a single core CPU, an application demanding performance will get full performance, as long as it is under TDP.

In the case that is more that one core, a multi-threaded workload could demand full performance on all cores. Thus the CPU is using the maximum frequency of the cores, that will consume more power than TDP. To avoid this, the CPU uses different maximum frequencies, depending on the number of active cores. So, if one core is more active than the other, it can use the remaining thermal budget and run at a higher frequency. To achieve this process INTEL introduced the Turbo Mode.

Figure 4.2: Visualization of TDP in a double Core CPU

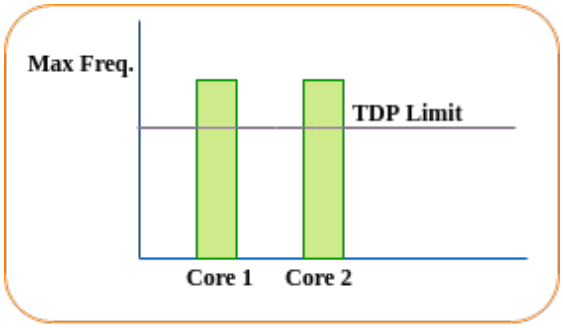
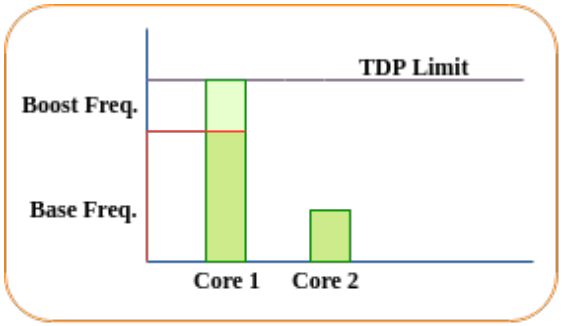


Figure 4.3: Visualization of TDP in a double Core CPU with Turbo Boost technology



As shown the Figure 4.2 when multi-threaded workload demand full performance on all cores the TDP limit is not considered. Opposite case as shown the Figure 4.3 when the boost technology is applied.

The Turbo Boost technology allows processor cores to run faster if is necessary than their base frequency, as long as the operating condition permits. Decide if is necessary allow processor cores to run faster the Power Control Unit (PCU) firmware is based on the:

- Number of cores active
- Estimated current consumption
- Estimated power consumption
- Processor temperature

Internal models and counters are used by the PCU in order to predict the actual and estimated power consumption. Since the Sandy Bridge microarchitecture added on-board power meter capability, which can be used to make better decisions. The calculations of power meters and power limits are exported through a set of MSRs (Machine Specific Registers) and PCIe config space. This interface is called the RAPL interface [18].

4.2 RAPL

Running Average Power Limit (RAPL) is a Linux kernel driver for the platforms INTEL, since Sandy Bridge microarchitecture. RAPL is a set of low-level interfaces with the ability to monitor, control, and get notifications of energy and power consumption data of different hardware levels [25]. RAPL is available from the Linux kernel 3.13.

RAPL is not an analog power meter, but rather uses a software power model, that estimates energy usage by using hardware performance counters and I/O models, implement a set of counters providing energy and power consumption information.

The information on energy consumption are stored in Machine-Specific Registers (MSRs), the same as those can be accessed by Operation System, such as the MSR kernel module in Linux. One advantage of RAPL is the energy/power consumption to be reported at a fine-grained manner like monitoring Package, CPU core, CPU uncore, and DRAM separately.

- Package.- Power consumption of all cores + LLC cache (last level cache)
- CPU Core .- Power consumption of all cores on socket
- CPU Uncore .- The RAPL PMU is uncore by nature and is implemented such that it only works in system-wide mode. Measuring only one CPU per socket is sufficient.
- DRAM.- Power consumption of DRAM

To fine grained reports and control the platforms are divided into domains, the domain is a physically meaningful domain for power management. The specific RAPL domains available in a platform vary across product segments.

Each RAPL domain supports [18]:

- ENERGY_STATUS for power monitoring.
- POWER_LIMIT and TIME_WINDOW for controlling power.
- PERF_STATUS for monitoring the performance impact of the power limit.
- RAPL_INFO contains information on measurement units, the minimum and maximum power supported by the domain.

4.3 Read_Rapl

Read_Rapl, is a C software based on RAPL. The software read the MSR files before and after a period of time, to get the information about the energy consumption. In this period of time, any software can be measure. To create a new software based on Read_Rapl is necessary to understand the software and then be available to modify it. The following sections are explained about those topics.

4.3.1 Understanding Read_Rapl

Firstly is fundamental understand the characteristics and permissions that the computer has to have in order to use the software. To read RAPL register, exists three ways to do that. Using MSRs, perf_event_open() interface and reading the values from sysfs powercap interface.

1. **Read the MSRs directly with /dev/cpu/"num_of_cpu"/msr.-** The driver must be enabled, and permissions set to allow read access. In some case might need to "modprobe msr" before it will work.
2. **Use the perf_event_open() interface.-** It requires at least Linux 3.14.
3. **Read the values from the sysfs powercap interface.-** It requires at least Linux 3.14.

Functions implemented

The principal functions implements are:

- **static int detect_cpu(void).-** The function detects the model of the computer. The INTEL architectures that the software support are:

  - Sandybridge
  - Ivybridge
  - Ivybridge-EP
  - Haswell
  - Haswell-EP
  - Broadwell

- **static int detect_packages(void).-** The function detects the numbers of cores and packages that the computer has.

The software reads in three different ways the RAPL register:

- Reading the MSR.
- Using perf_event_open.
- Using the sysfs powercap interface.

The functions implement for each way are:

- **Reading the MSRs.-**
  - **open_msr.-** Simply open the file where is written the energy power consumption.
  - **read_msr.-** Return the data or information that was read in the files MSR.
  - **rapl_msr.-** The function read the amount of energy consumption before and after of the determinate period time, and show the energy consumption in that period.
- **Using the perf_event_open() interface.-**
  - **perf_event_open .-** Open the files MSR using the event syscall.
  - **rapl_perf.-** The function store the information obtained using the event perf_event_open.
  - **rapl_domain_names.-** Define the names of the domains: Package,Core,Uncore and Dram.
- **Reading the values from the sysfs powercap interface.-**
  - rapl_sysfs.- Using the sysfs powercap interface, gather results. Store the information in two matrices before and after the period of time to will be measured.

```
1 long long before[MAX_PACKAGES][NUM_RAPL_DOMAINS];
2 long long after[MAX_PACKAGES][NUM_RAPL_DOMAINS];
```

4.3.2 Modifying the Read_Rapl software

The previous section presented three ways how Read_Rapl reads the energy consumption from the MSR files. To this Thesis two ways will be modified. The following are the functions that have been modified.

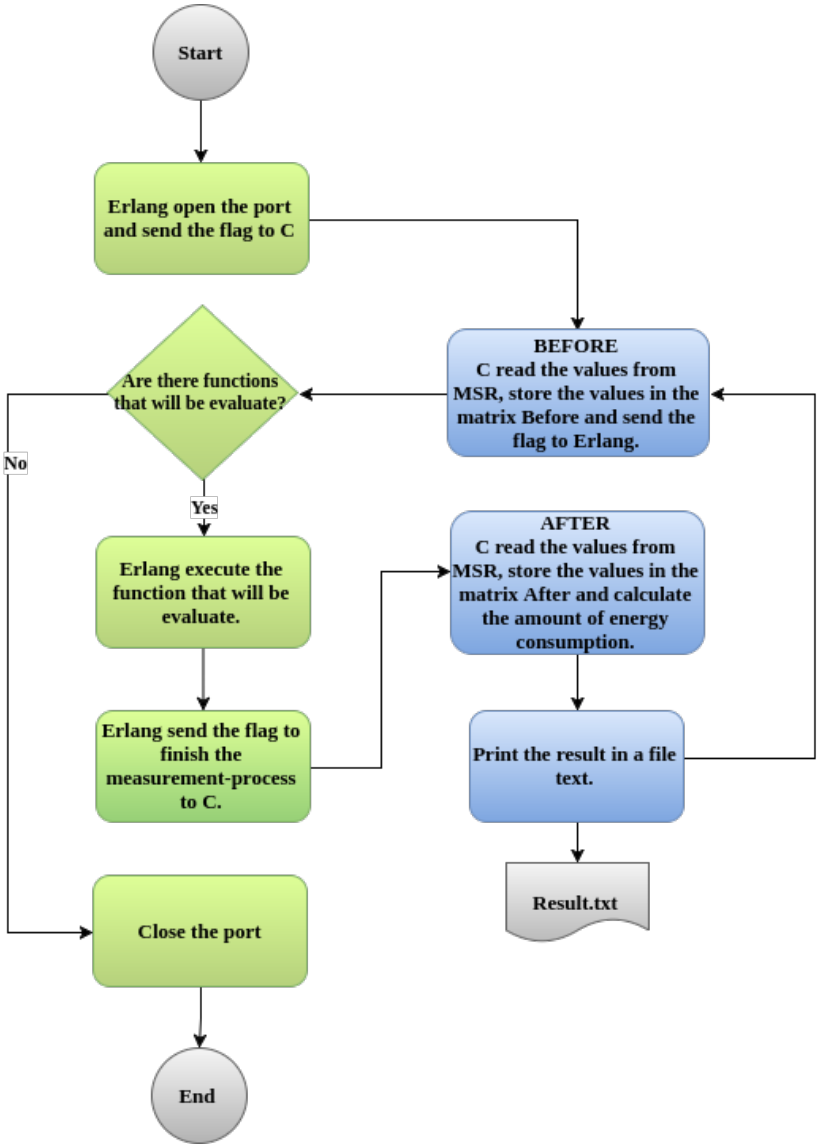
- **Trying with sysfs powercap interface.-** In the original file the measured of the energy consumption is performed in a single function. The adaptation is performed instead of one single function split into two functions, one read the energy consumption before and the other one after, in such way that between them the Erlang function is executed. The functions are:
  - rapl_sysfs_before.- The function reads the amount of the energy consumption before the Erlang function is executed, and store those values in the respective matrix "before".
  - rapl_sysfs_after.- The function reads the amount of the energy consumption after the Erlang function is executed and store those values in the respective matrix "after". Then the subtraction is performed, between the matrix after and before getting the final result.
- **Trying with perf_event interface.-** In the original file the measurement of the energy consumption is performed in a single function. The adaptation is performed instead of one single function split into two functions, between the two functions the Erlang function is executed. The functions are:
  - rapl_perf_before.- The function reads the amount of the energy consumption using the event perf_event_open() and store in the matrix called fd, before the Erlang function is executed.
  - rapl_perf_after.- The function reads the amount of the energy consumption using the event perf_event_open() after the Erlang function is executed, then the subtraction is execute between after and before the amount the energy consumption is assigned to a variable.
- **Main.-** In this function after the software detected which core, the number of packages and if the System operation has the requirements required, call the function rapl_sysfs_before and rapl_perf_before. Then the Erlang function is called and executed. Then when the execution finish, the functions rapl_sysfs_after and rapl_perf_after are executed. The calculation of the amount of energy consumption is performed between the matrix after and before and the results are shown in four domains: Package, Core, Uncore and Dram. The amount of energy is given in Joules.

4.4 Joining with Erlang

In the adaptation of the software Read_Rapl between the function before and after the Erlang functions has to be executed. Read_Rapl is coded in C and, on the side of Erlang, ports [26]

are using to achieve the connection between Erlang and C. The behavior between Erlang and C is sending and receiving values like flags. Depending on the values Read_Rapl reads the values from the MSRs, or in the side of Erlang the functions are executed. Finally, the results are stored in a flat file.

Figure 4.4: Flow chart of the communication between Erlang functions and Read_Rapl

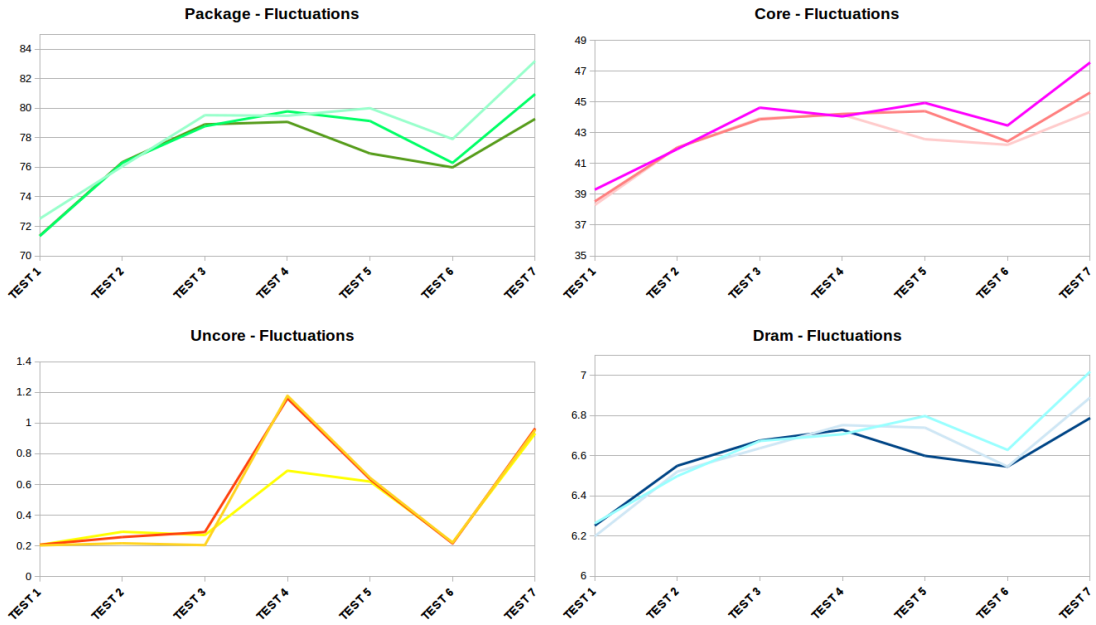


As shown in Figure 4.4 the communication between C and Erlang is working based on flags either in C or Erlang side. Depending on the value that receives read the MSR files or execute the Erlang functions. Finally, all the results are store in the file `result.txt`.

4.5 Methodology to measure the energy consumption in Erlang

The thesis is focused in the measure of energy consumption and, one of the critical points to be considered is the methodology that is going to be used. The behavior of the computer depends on many factors, the energy that the computer consume could be affected and show wrong results. The tests in the beginning, show that the fluctuation was a problem.

Figure 4.5: Fluctuations per variable



As shown in Figure 4.5 the same test was executed seven times and show that exist picks either bigger amount or lower amount of energy consumption in all variables.

Analyzing that scenario to avoid this kind of problems the following methodology have been implemented:

- The computer must have the same conditions and with the lowest possible consumption per each test. The lowest possible consumption means no use resources like video, audio, etc.
- Every test has been executed twelve times.
- To avoid the fluctuation the lowest and highest value has been left from the scenario.
- Finally the amount of energy consumption is the average of ten values.

- Due to the fact that per each variable the amount of energy in Joules are different comparing one with another, the percentage of consumption are considered to investigate on the presented figures.

4.6 Test environment

The next chapter contains all the experiments have been realized. Is important remark the characteristics of the computer that has been used for all the test.

- **Physical characteristics.-**
    1. **Brand.-** Toshiba Satellite
    2. **Model.-** L50
    3. **Procesor.-** Intel Core i7-4700MQ 2.4 Ghz
    4. **Microarchitecture.-** Haswell
    5. **Memory.-** 12 Gigas
- **Operating system.-** Linux
    1. **Distribution.-** Ubuntu
    2. **Version.-** 16.04 LTS

Chapter 5

Experiments

The aim of this chapter is measured the energy consumption of Erlang functions, solving the same problem by implementing different algorithms or performing some refactorings on them. To test if the amount of energy change depending on those modifications. The software used is Read_RAPL join with Erlang, explained in the previous chapter. In the end of this chapter there are the section Discussion where are the results of those tests. The target is fond some patterns or recommendations to reduce the energy consumption in Erlang functions.

The result to each tests is shown in the next variables:

1. **Package.-** Power consumption of all cores + LLC cache (last level cache).
2. **CPU Core. -** Power consumption of all cores on a socket.
3. **CPU Uncore.-** The RAPL Power Management Unit (PMU) is uncore by nature and is implemented such that it only works in system-wide mode. Measuring only one CPU per socket is sufficient.
4. **DRAM.-** Power consumption of DRAM.

It is important to highlight that the unit of measure of the variables is JOULES, and the methodology used is the one described in the previous chapter.

5.1 Different algorithms

Create different algorithms from the same function is a kind of refactoring. To this section the following functions `Sum List`, `Fibonacci` and `Karatsuba` created with different algorithms, has been measured.

5.1.1 Sum List

The sum is the addition of a sequence of numbers and the result is their sum or total, in this case, the sequence of numbers is given in an List [27]. The different algorithms have been implemented are:

- `Primitive recursive Function`
- `Tail recursive Function`
- `Introducing variables`
- `Higher order Function`

Source code

The following are the source codes of the algorithms that have been implemented.

- **Primitive recursive Function.-** As shown in Figure 5.1 the list ([H|T]) is performed analyzing the head and sending recursively the tail.

Figure 5.1: Sum List - Primitive recursive Function

```
1 sum_recursion([]) ->
2         0;
3 sum_recursion([H|T]) ->
4         H+sum_recursion(T).
```

- **Tail recursive.-** As shown in Figure 5.2 the variable `ACC` is used like accumulator.

Figure 5.2: Sum List - Tail recursive function

```
1 sum_acc([H|T]) ->
2         sum_acc_help([H|T],0).
3
4 sum_acc_help([H|T],Acc)->
5         sum_acc_help(T,Acc+H);
6
7 sum_acc_help([],Acc) ->
8         Acc.
```

- **Introducing Variables.-** As shown the Figure 5.3 the variable X is used to stored the Acc and evaluate the sum using `Tail recursive`.

Figure 5.3: Sum List - Introducing of Variables

```
sum_var([H|T]) ->
        sum_var_help([H|T],0).

sum_var_help([H|T],Acc)->
        X=Acc+H,
        sum_acc_help(T,X);

sum_var_help([],Acc) ->
        Acc.
```

- **Higher order Function.-** As shown in Figure 5.4 the function `foldl` is used to evaluate the sum.

Figure 5.4: Sum List - Higher order Function

```
sum_hod(L) ->
 lists:foldl(fun(X, Sum) -> X + Sum end, 0, L).
```

- **Value to evaluate.-** The list `lists:seq(10,50000000)` has been evaluated.
- **Results.-** The `Primitive recursive Function` is taken as Base Case in such way that each algorithm is compared with the base case.

Table 5.1: Amount of energy consumption

	Package	**Core**	**Uncore**	**Dram**	**SUM (Joules)**
Primitive Recursive Function (Base Case)	15.96	8.31	0.11	2.41	26.78
Tail Recursive Function	5.46	2.72	0.09	0.58	8.85
Introduction of Variables	5.38	2.68	0.08	0.58	8.72
Higher order Function	18.18	9.98	0.12	1.64	29.92

Table 5.2: Primitive recursive Function VS All others algorithms - Percentage

	Package	**Core**	**Uncore**	**Dram**	**SUM**
Primitive recursive Function (Base Case)	100.00%	100.00%	100.00%	100.00%	100.00%
Tail recursive function	34.20%	32.76%	80.54%	24.08%	33.03%
Introducing variables	33.70%	32.24%	78.00%	24.07%	32.56%
Higher order Functions	113.94%	120.09%	109.36%	68.17%	111.72%

Figure 5.5: Primitive recursive Function VS All others algorithms - Variables

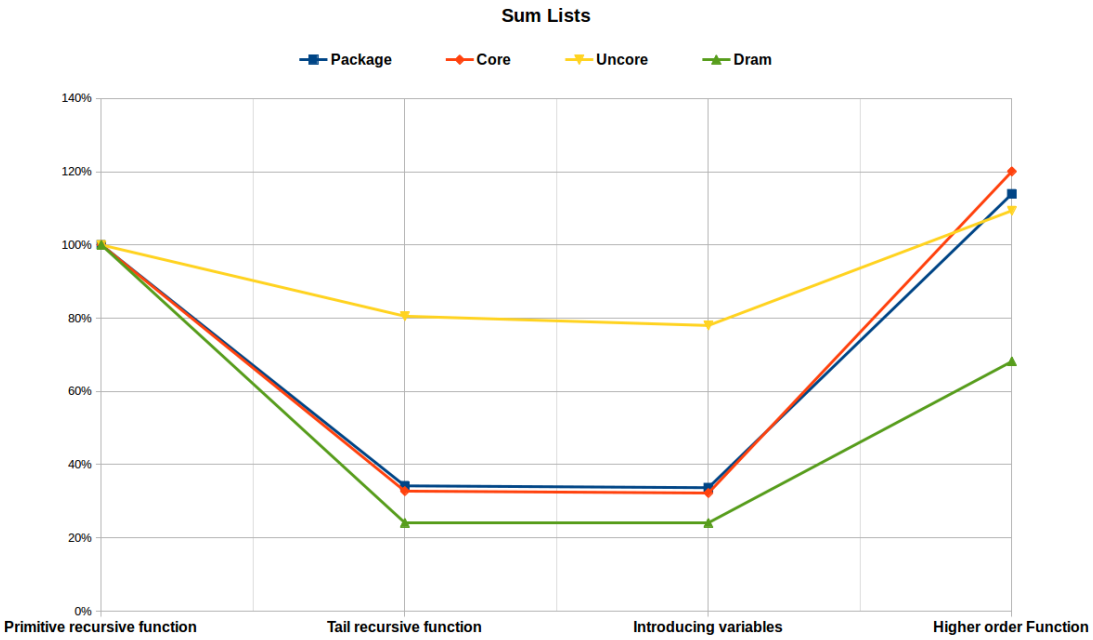
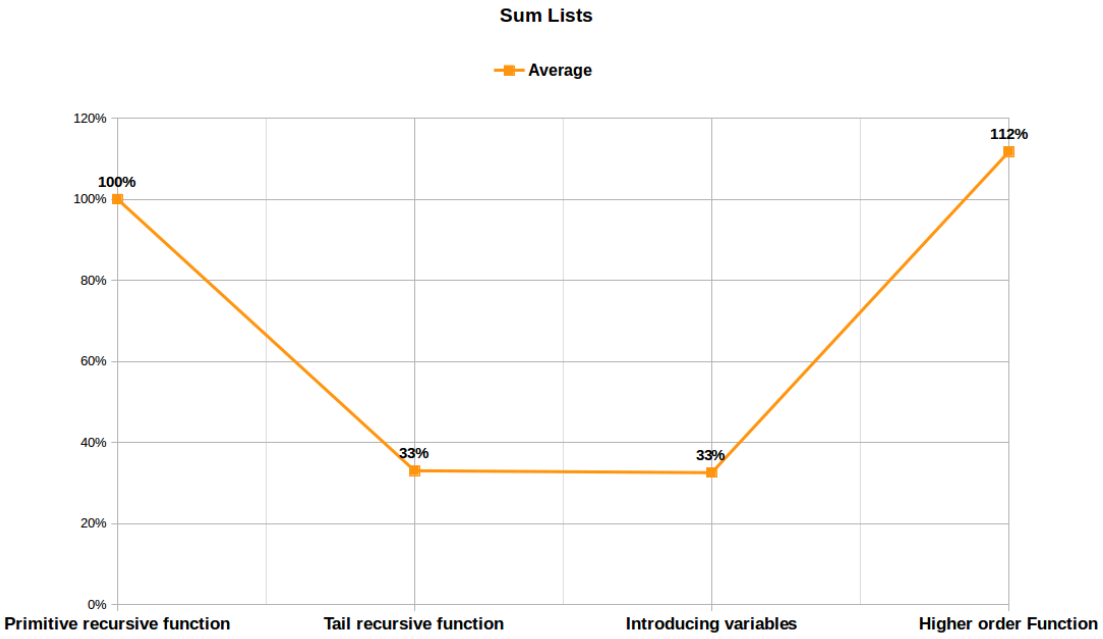


Figure 5.6: Primitive recursive Function VS All others algorithms - Average



- **Analysis.-** It can be observed that `Primitive recursive Function` comparing with each refactoring, the `Tail recursive function` consume less energy as shown in Figure 5.12. Opposite case `Higher order Functions` consume more energy in almost all variables as shown in Figure 5.5. The Table 5.1 presents the amount in Joules per function and, per variable in Table 5.2. The percentage are visualized in Figure 5.12. From this tests is grained the following tests to determine the amount of efficiency in specific cases:

  1. Primitive recursive Function vs Higher order Function.
  2. Tail recursive Function vs Higher order Function.
  3. Primitive recursive Function vs Tail recursive Function.
  4. Tail recursive Function vs Introducing Variables.

Primitive recursive Function vs Higher order Function

The Figure 5.12 show that the worst algorithm was `Higher order Function` compared with the Base case `Primitive recursive Function`. This test has been created to determine the amount of energy that is saved using `Primitive recursive Function` against `Higher order Function`.

- **Value to evaluate.-** The list `lists:seq(10,50000000)` has been evaluated.

Table 5.3: Percentage reduction or increase

	Package	Core	Uncore	Dram	SUM
Primitive recursive function	100.00%	100.00%	100.00%	100.00%	100.00%
Higher order Function	113.94%	120.09%	109.36%	68.17%	111.72%

Figure 5.7: Percentage of the energy consumption per variables

- **Analysis.-** The results on the comparison between both implementations are presented in Figure 5.7. `Higher order Function` consume more energy than `Primitive recursive Function` in almost all variables. Is important remark as shown in Table 5.7 the variable `Dram` with `Higher order Functions` the energy consumption decrease around 32%. The amount in Joules is shown in Table 5.1.

Tail recursive Function vs Higher order Function

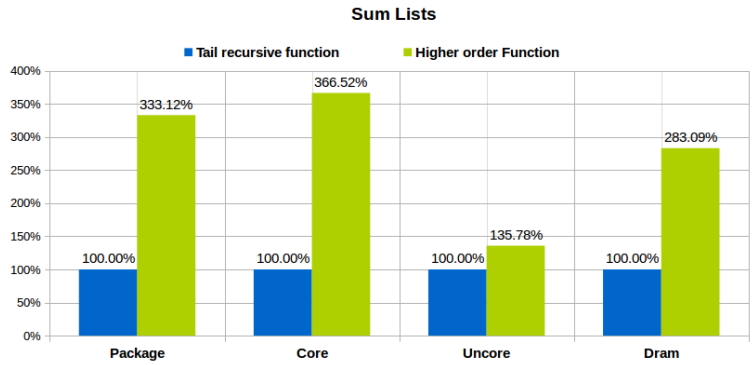
The Figure 5.12 show that the worst algorithm was `Higher order Function` and the best algorithm was `Tail recursive Function`. This test has been created to determine the amount of energy that is saved using `Tail recursive Function` against `Higher order Function`.

- **Value to evaluate.-** The list `lists:seq(10,50000000)` has been evaluated.
- **Result.-**

Table 5.4: Percentage reduction or increase

	Package	Core	Uncore	Dram	SUM
Tail recursive function	100.00%	100.00%	100.00%	100.00%	100.00%
Higher order Functions	333.12%	366.52%	135.78%	283.09%	338.21%

Figure 5.8: Percentage of the energy consumption per variables



- **Analysis** .- The results on the comparison between both implementations are presented in Figure 5.8. `Tail recursive function` consume less energy than `Higher order Function`. It is important remark that the amount of energy of variable `Core` in `Higher order Functions`, is three times the amount comparing with `Tail recursive function`, as shown in Table 5.4. The amount in Joules is shown in Table 5.1.

Primitive recursive Function vs Tail recursive Function

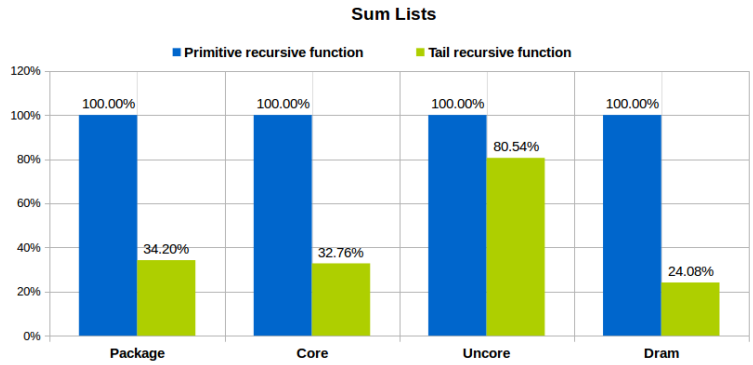
In Erlang the recursive functions are useful. The test `Sum List` has been created two algorithms based on recursive, `Primitive` and `Tail recursive`. This test has been created to determine which algorithms consume less energy.

- **Value to evaluate.-** The list `lists:seq(10,50000000)` has been evaluated.

Table 5.5: Percentage reduction or increase

	Package	**Core**	**Uncore**	**Dram**	**SUM**
Primitive recursive function	100.00%	100.00%	100.00%	100.00%	100.00%
Tail recursive function	34.20%	32.76%	80.54%	24.08%	33.03%

Figure 5.9: Percentage of the energy consumption per variables



- **Analysis.-** The results on the comparison between both implementations are presented in Figure 5.9. `Tail recursive function` consume less energy than the `Primitive recursive`. In variable like `Package` consume three times less than in `Primitive recursion` as shown in Table 5.5. The amount in Joules is shown in Table 5.1.

Tail recursive Function vs Introducing Variables

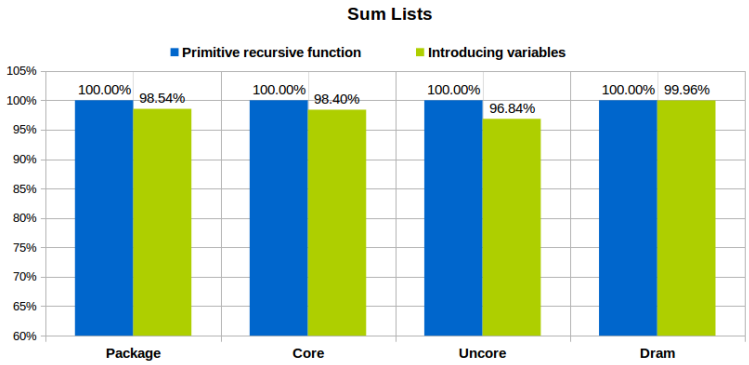
A typical case of refactoring is the Introduction of Variables. This test has been created to analyze if the `Introduction of Variables` affect the amount of energy consumption.

- **Value to evaluate.-** The list `lists:seq(10,50000000)` has been evaluated.

Table 5.6: Percentage reduction or increase

	Package	**Core**	**Uncore**	**Dram**	**SUM**
Primitive recursive function	100.00%	100.00%	100.00%	100.00%	100.00%
Tail recursive function	34.20%	32.76%	80.54%	24.08%	98.57%

Figure 5.10: Percentage of the energy consumption per variables



- **Analysis** .- The results on the comparison between both implementations are presented in Figure 5.10. The energy consumption decreased in all variables, but it was a small amount. Analyzing the Table 5.6 the maximum reduction is in the variable `Uncore` with around 3% of decrease. The reduction is insignificant, so far the `Introducing Variables` does not affect the energy consumption. The amount in Joules is shown in Table 5.1.

5.1.2 Algorithms to Fibonacci series

The Fibonacci numbers are the numbers in the following integer sequence, called the Fibonacci sequence, and characterized by the fact that every number after the first two is the sum of the two preceding ones [28].

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987$$

Often, especially in modern usage, the sequence is extended by one more initial term:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987$$

To this test, the Fibonacci series has been coded in different techniques:

1. Array style.
2. List structure.
3. List reverse order.
4. Minimal data structure.

Source code

To this kind of experiments is important analyze the source code. To this test the codes have been attached in the Appendix A. The Appendix contain the four algorithms to fibonnaci series:

1. Array style.- The function use the package `array` to evaluate the fibonacci series and store the value in an array.
2. List structure.- The function use the package `lists` to evaluate the fibonacci series and store the value in a list.
3. List reverse order.- The function is using lists and evaluating in reverse order.
4. Minimal data structure.- The function use just three lines of implementation and just integer resources.

- **Case to test. -** The number `50000` has been evaluated.
- **Result.-** The `Array style` is taken as Base Case in such way that each algorithm is compared with the base case.

Table 5.7: Amount of energy consumption

	Package	**Core**	**Uncore**	**Dram**	**SUM (Joules)**
Array style	2.57	1.36	0.04	0.38	4.36
List structure	508.46	286.44	1.07	49.27	845.24
List reverse order	215.13	101.05	0.78	22.81	339.77
Minimal data structure	0.53	0.27	0.03	0.05	0.88

Table 5.8: Array style vs all other algorithms - Percentage

	Package	**Core**	**Uncore**	**Dram**	**SUM**
Array style	16.12%	16.40%	41.64%	15.86%	100.00%
List structure	3186.07%	3446.28%	998.29%	2048.13%	19378.59%
List reverse order	1348.03%	1215.77%	729.42%	948.28%	7789.88%
Minimal data structure	3.32%	3.25%	29.68%	2.20%	20.28%

Figure 5.11: Array style VS all other algorithms - Variables

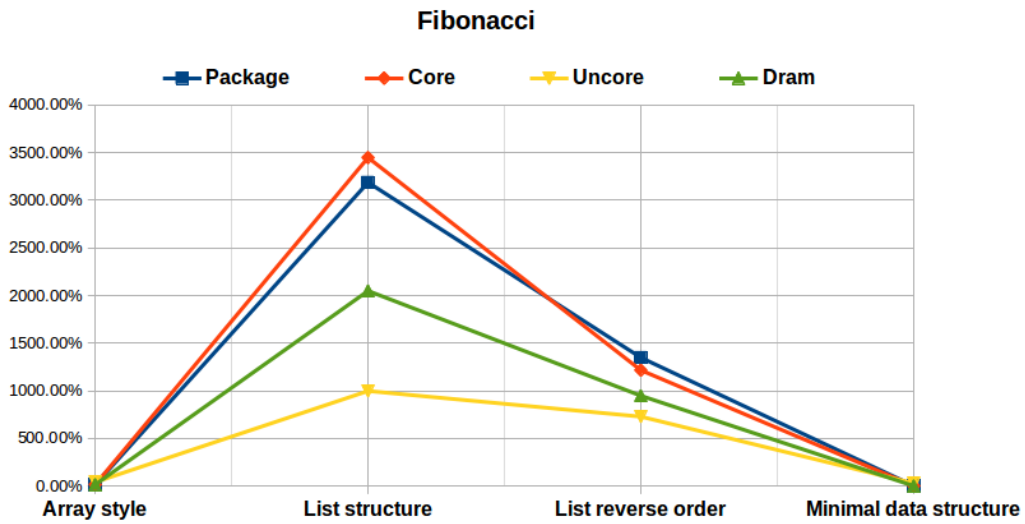
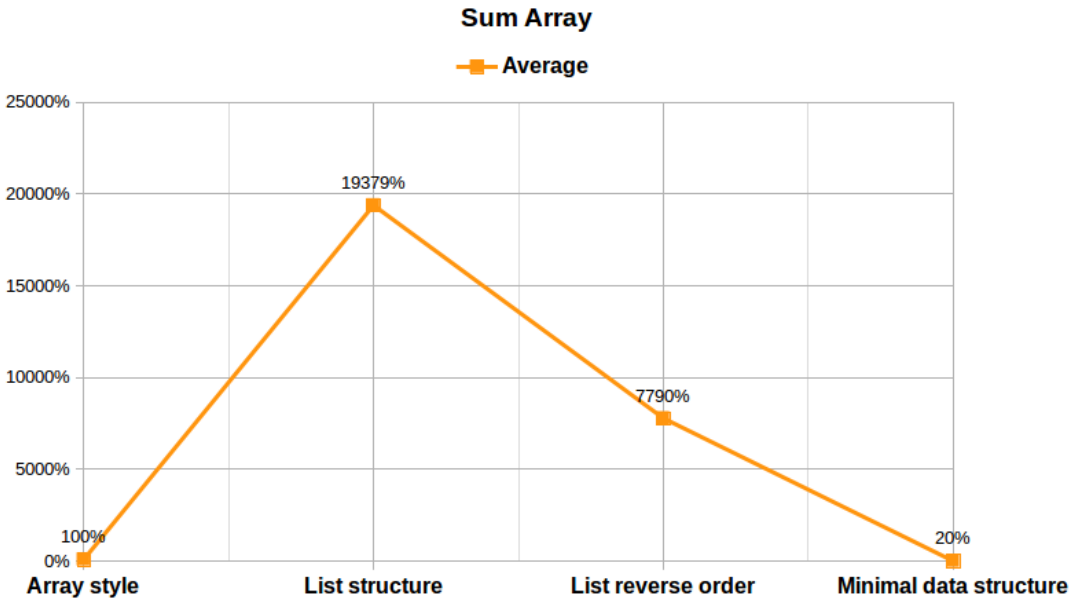


Figure 5.12: Array style VS all other algorithms - Average



- **Analysis**.-It can be observed that `Array style` the Base case comparing with each refactoring, `List structure` has the higher energy consumption as shown in Figure 5.12. The

most efficient algorithms are `Array style` and `Minimal data Structure` but is not possible to determine which is the best algorithm. For this reason and to determine the amount of efficiency in specific cases the following case are grained:

1. Array-style vs Minimal data Structure.
2. Array-style vs Lists structure.

Array-style vs Minimal data Structure

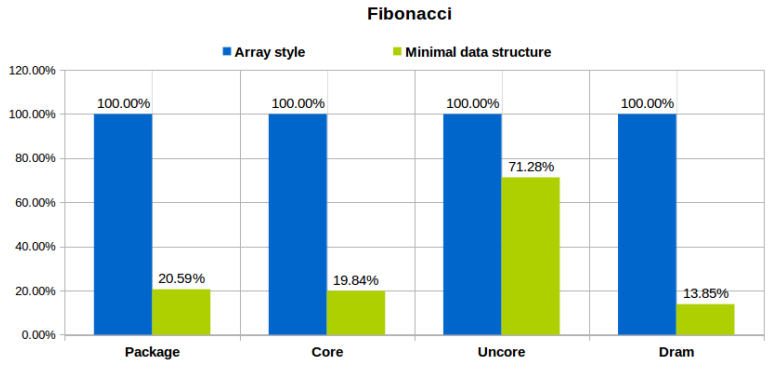
In Figure 5.12 show that the more efficient algorithms were `Array-style` and `Minimal data Structure`. This test has been created to determine which algorithm between them is more efficient.

- **Case to test. -** The number `50000` has been evaluated.
- **Result.-**

Table 5.9: Percentage reduction or increase

	Package	Core	Uncore	Dram	SUM
Array style	100.00%	100.00%	100.00%	100.00%	100.00%
Minimal data structure	20.59%	19.84%	71.28%	13.85%	20.28%

Figure 5.13: Percentage of the energy consumption per variables



- **Analysis.-** The results on the comparison between both implementations are presented in Figure 5.13. `Minimal data Structure` reduce the energy consumption in all variables, at less 80% as shown in Table 5.9. The amount in Joules is shown in Table 5.7.

Array-style vs Lists structure

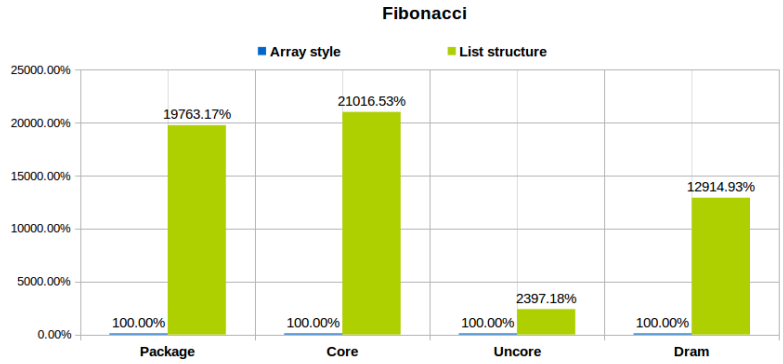
`Array` and `List` are two common typical algorithms used in Functional programming. Based on `Algorithm to Fibonacci` this test has been created to determinate the amount of increment in the energy consumption when Lists are applying.

- **Case to test.-** The number `50000` has been evaluated to this test.
- **Result.-**

Table 5.10: Percentage reduction or increase

	Package	Core	Uncore	Dram	SUM
Array style	100.00%	100.00%	100.00%	100.00%	100.00%
List structure	19763.17%	21016.53%	2397.18%	12914.93%	19378.59%

Figure 5.14: Percentage of the energy consumption per variables



- **Analysis.-** The results on the comparison between both implementations are present in Figure 5.13. The increment of energy consumption using list is huge, as shown in Table 5.10. Definitely, the lists are not the more efficient algorithm when the target is reduce the energy consumption. The amount in Joules is shown in Table 5.7.

Lists structure vs List reverse order

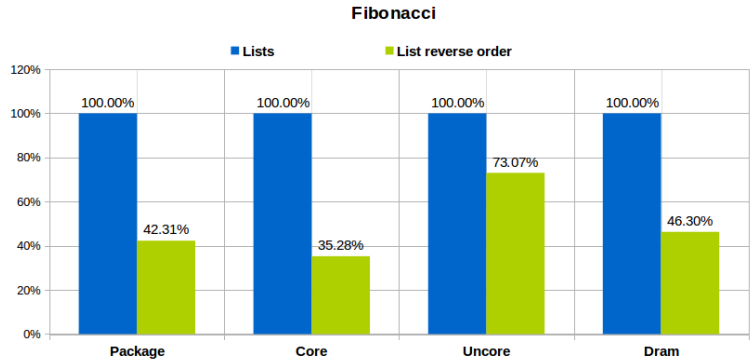
`Lists` always are useful in Functional programming. This test has been created to determinate which algorithm is more efficient between `List` and `Reverse Lists`.

- **Case to test.-** The number `50000` has been evaluated to this test.
- **Result.-**

Table 5.11: Percentage reduction or increase

	Package	Core	Uncore	Dram	Sum
Lists	100.00%	100.00%	100.00%	100.00%	100.00%
List reverse order	42.31%	35.28%	73.07%	46.30%	40.20%

Figure 5.15: Percentage of the energy consumption per variables



- **Analysis.-** The results on the comparison between both implementations are present in Figure 5.15. The reduction of energy consumption with `Reverse List` in average is 60% as show in Table 5.11. The amount in Joules is shown in Table 5.7.

5.1.3 Karatsuba Binary and Lists

The Karatsuba algorithm is a fast multiplication algorithm. It reduces the multiplication of two `n-digit` numbers to at most $n^{\log_2 3} \approx n^{1.585}$ single-digit multiplications. It is therefore faster than the classical algorithm, which requires n^2 single-digit products. For example, the Karatsuba algorithm requires $3^{10} = 59{,}049$ single-digit multiplications to multiply two 1024-digit numbers ($n = 2^{10} = 1024$), whereas the classical algorithm requires $(2^{10})^2 = 1,048,576$ [29].

Two algorithms have been created in order to analyze:

- Binary
- Lists

Source code

To this kind of experiments is important analyze the source code. To this test the codes have been attached in the Appendix B.

- Binary.- This algorithm is using a binary to represent the numbers and the evaluation is in binary way.

- Lists.- This algorithm is using a list to represent the numbers and the evaluation is with integer numbers.

- **Case to test. -** The numbers have been evaluated are `123456789 161 times (*) 123456789 161 times`.

- **Results.-**

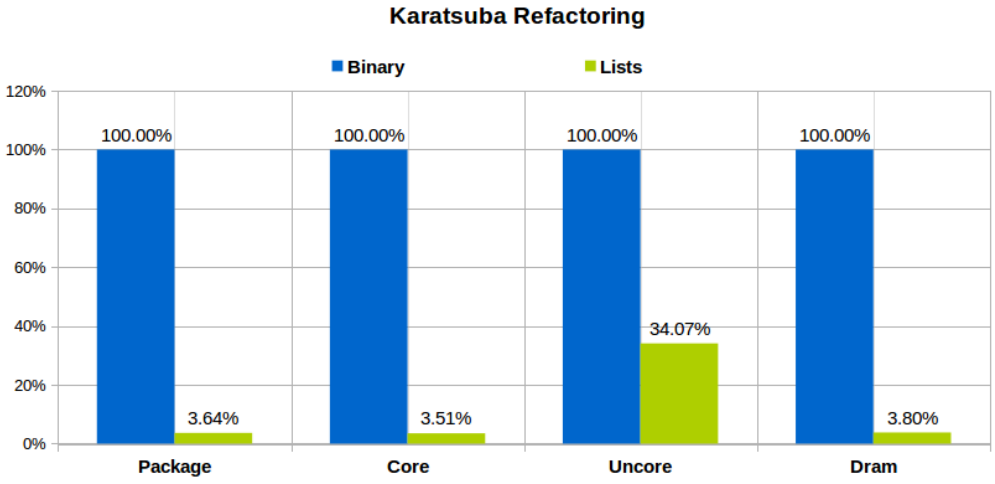
Table 5.12: Amount of energy consumption

	Package	**Core**	**Uncore**	**Dram**	**SUM (Joules)**
Binary	64.43	37.88	0.21	5.36	107.88
Lists	2.35	1.33	0.07	0.20	3.95

Table 5.13: Percentage reduction or increase

	Package	**Core**	**Uncore**	**Dram**	**SUM**
Binary	100.00%	100.00%	100.00%	100.00%	100.00%
Lists	3.64%	3.51%	34.07%	3.80%	3.67%

Figure 5.16: Percentage of the energy consumption per variables



- **Analysis**.- The results on the comparison between both implementations are presented in Figure 5.16. Both cases are using a list, but Binary is inefficiency. As shown in Figure 5.16 the simple list is 30 times more efficient. Simple List just consumes 3.67% of energy compared with the Binary algorithm, as shown in Table 5.13.

5.2 Refactoring Tests

Refactoring is the process where the developers restructure existing computer without changing its external behavior. The functions `Fibonacci` and `Karatsuba` have been refactored in different ways, in order to determine which refactoring save more energy.

5.2.1 Fibonacci Refactoring

The section 5.1.2 have the description of Fibonacci. The Fibonacci series have been refactored [30] seven times, the following are the transformations:

1. Recursive version.
2. Introducing variables.
3. Binding List.
4. Higher order Function.
5. Introducing variables - Higher order Function.
6. Introducing variables - Higher order Function v2.
7. Parallel version.

Source code

Like the previous tests is important to analyze the source code. The codes to this test are attached in the Appendix C. The following are the codes:

1. Recursive version.- Using the primitive recursive version to get the result.
2. Introducing variables.- Base in the recursive version, this refactored introduce the variables `A and B`.
3. Binding List.- The algorithm apply Binding to List refactoring.
4. Higher order Function.- The algorithm introduce Higher order function (`lists:map`)
5. Introducing variables - Higher order Function.- Based on the previous algorithm introduce the variable `SubPr`.
6. Introducing variables - Higher order Function v2.- Based on the previous algorithm introduce the variable `SubSols`.
7. Parallel version.- Base in the Higher order function version, use the function `pmap`.

- **Case to test.-** The number 40 has been evaluated.

- **Result.-** The `Recursive function` is taken as Base Case in such way that each algorithm is compared with the base case.

Table 5.14: Amount of energy consumption

	Package	**Core**	**Uncore**	**Dram**	**SUM (Joules)**
Recursive version	83.21	48.65	0.22	6.63	138.71
Introducing variables	75.57	41.21	0.20	6.43	123.41
Binding List	75.91	41.94	0.21	6.53	124.60
Higher order Function	344.21	192.18	0.51	28.96	565.85
Introducing variables – HOF	348.98	193.85	0.54	29.42	572.79
Introducing variables – HOF V2	358.93	199.50	0.55	30.18	589.16
Parallel version	65.95	42.49	0.15	4.54	113.13

Table 5.15: Recursive version vs all other refactoring - Percentage

	Package	**Core**	**Uncore**	**Dram**	**SUM**
Recursive version	521.42%	585.28%	202.53%	275.63%	585.28%
Introducing variables	473.52%	495.85%	185.53%	267.30%	473.52%
Binding List	475.68%	504.62%	196.40%	271.52%	475.68%
Higher order Function	2156.84%	2312.14%	474.63%	1203.84%	2156.84%
Introducing variables – HOF	2087%	2232%	403%	1123%	1461%
Introducing variables – HOF V2	2149%	2300%	413%	1154%	1504%
Parallel version	313%	411%	38%	89%	213%

Figure 5.17: Recursive version vs all other refactoring - Variables

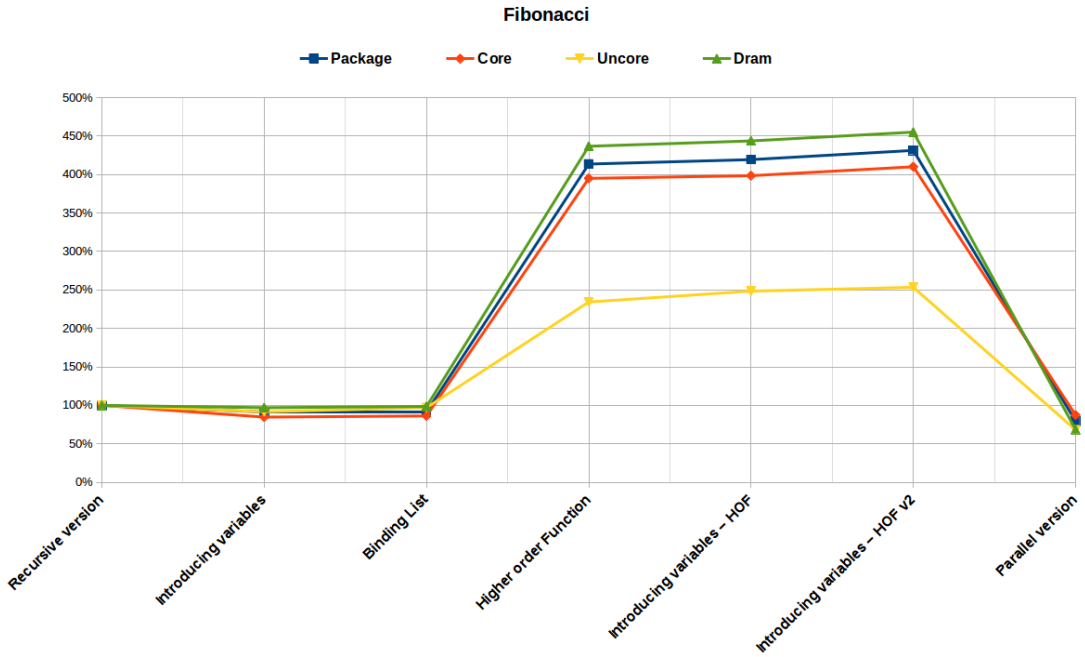
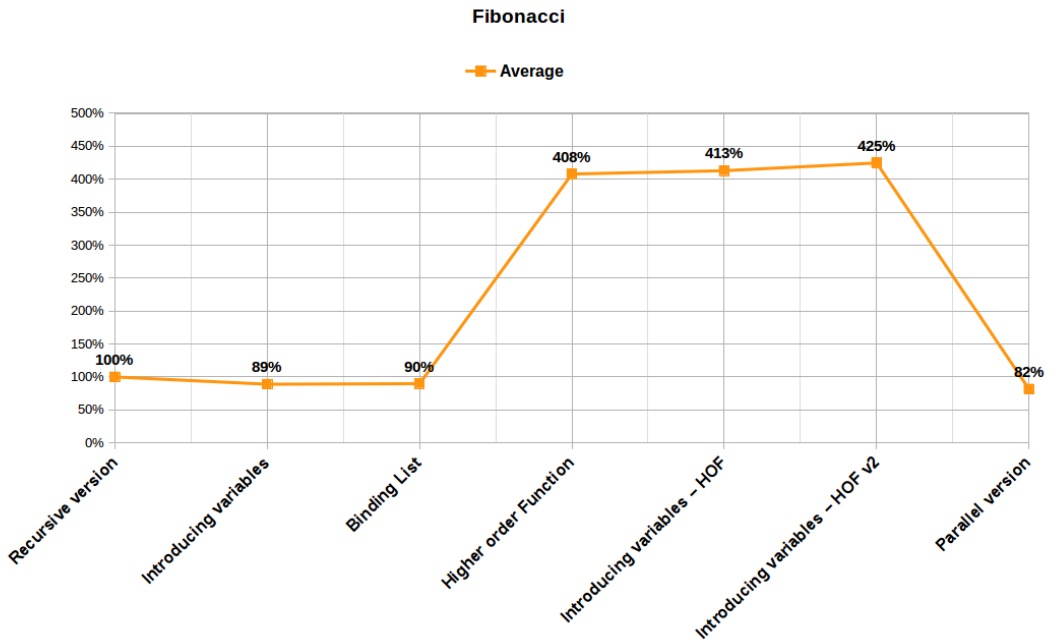


Figure 5.18: Recursive version vs all other refactoring - Average



- **Analysis**.- Considering that the `Recursive version` is the Base case, as shown in Figure 5.17 the first three keep similar the energy consumption to all variables, but increase when the `Higher Order Functions` are implemented. In the end of the implementation with Parallel version the energy consumption decrease again, even using `Higher order Functions`. Based on this the following cases have been grained, to determine the amount of efficiency in specific cases:

  1. Recursion version vs Higher order Functions.
  2. Recursion and Higher order Functions vs Introducing variables.
  3. Parallel version vs Higher order Function.
  4. Recursion version vs Parallel version.

Recursion version vs Higher order Function

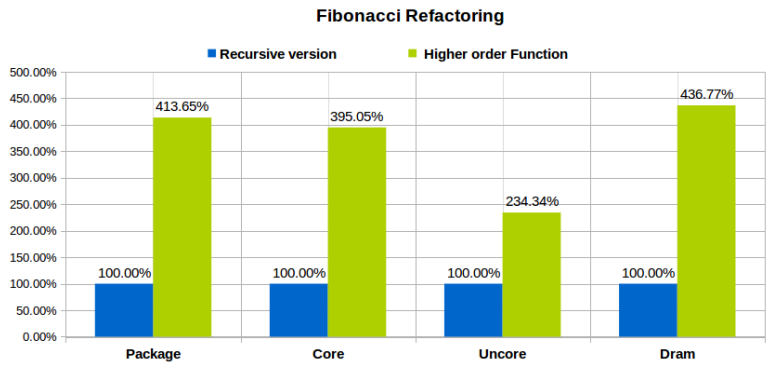
In Figure 5.18 present that the worst refactoring is `Higher order Functions`. This test has been created to know the increment of energy consumption when `Higher order Functions` is chosen against `Recursive version`.

- **Case to test.-** The number 40 has been evaluated.
- **Results.-**

Table 5.16: Percentage reduction or increase

	Package	Core	Uncore	Dram	SUM
Recursive version	100.00%	100.00%	100.00%	100.00%	100.00%
Higher order Function	413.65%	395.05%	234.34%	436.77%	407.95%

Figure 5.19: Percentage of the energy consumption per variables



- **Analysis**.- The results on the comparison between both implementations are presented in Figure 5.19. `Higher order Function` increase the energy consumption in all variables, on

average the increment is four times more than the `Recursive version`. The amount in Joules is shown in Table 5.14.

Introducing variables

The Introduction of variables is one of the more common refactoring that has been implemented. This test is going to compare two cases: `Introduction of variables` in the `Recursive version` and in `Higher order Functions` version.

- **Case to test.-** The number 40 has been evaluated.
- **Results.-**

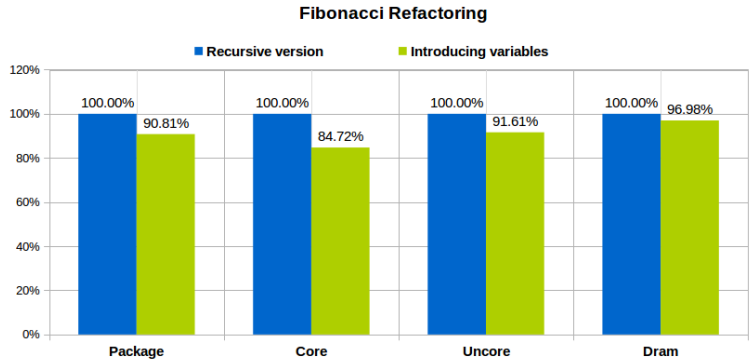
Table 5.17: Percentage reduction or increase

	Package	**Core**	**Uncore**	**Dram**	**SUM**
Recursive version	100.00%	100.00%	100.00%	100.00%	100.00%
Introducing variables	90.81%	84.72%	91.61%	96.98%	88.97%

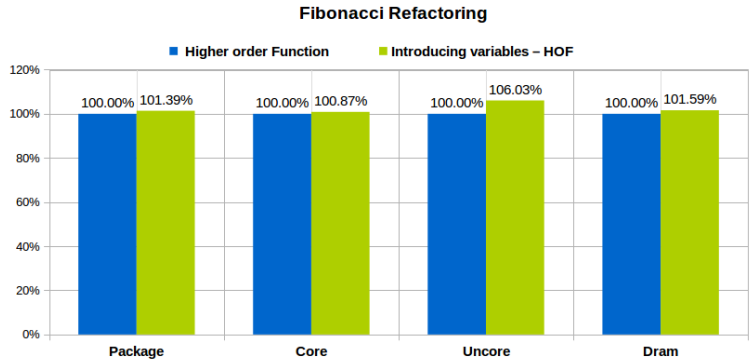
Table 5.18: Percentage reduction or increase

	Package	**Core**	**Uncore**	**Dram**	**SUM**
Higher order Function	100.00%	100.00%	100.00%	100.00%	100.00%
Introducing variables – HOF	101.39%	100.87%	106.03%	101.59%	101.23%

Figure 5.20: Percentage of the energy consumption per variables



(a) Introducing variables - Recursion version



(b) Introducing variables - Higher order Function

- **Analysis**.-The results on the comparison of implementations are presented in Figure 5.20. `Introduction of variables` in both cases have different behaviors. In the case of `Recursive version` the amount of energy decrease, but in `Higher order Function` increase. The amount of increment or decrease is not substantively amount, because as shown in Table 5.18 in one case decrease 11% and in the other just increase 1%. The amount in Joules is shown in Table 5.14.

Parallel version vs Higher order Function

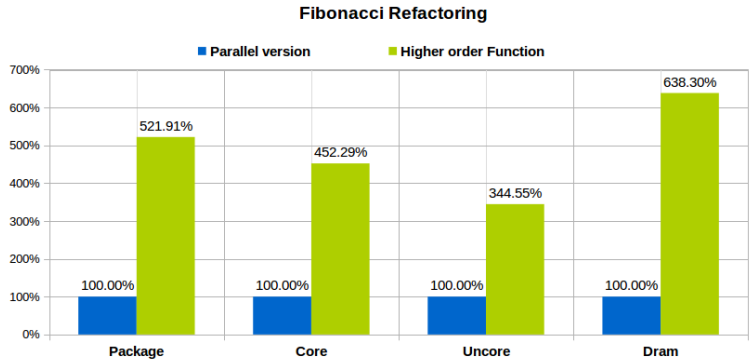
`Parallel version` and `Higher order Function` are function are functions useful in Programing Language. This test has been created to know which is more efficient base on `Fibonacci Refactoring` and the amount of efficiency.

- **Case to test.-** The number 40 has been evaluated.
- **Results.-**

Table 5.19: Percentage reduction or increase

	Package	Core	Uncore	Dram	SUM
Parallel version	100.00%	100.00%	100.00%	100.00%	100.00%
Higher order Function	521.91%	452.29%	344.55%	638.30%	500.20%

Figure 5.21: Percentage of the energy consumption per variables



- **Analysis**.- The results on the comparison between both implementations are presented in Figure 5.21. `Higher order Function` increase the energy consumption in all variables. On average the increment is five times more than `Parallel version`. The amount in Joules is shown in Table 5.14.

Recursive version vs Parallel version

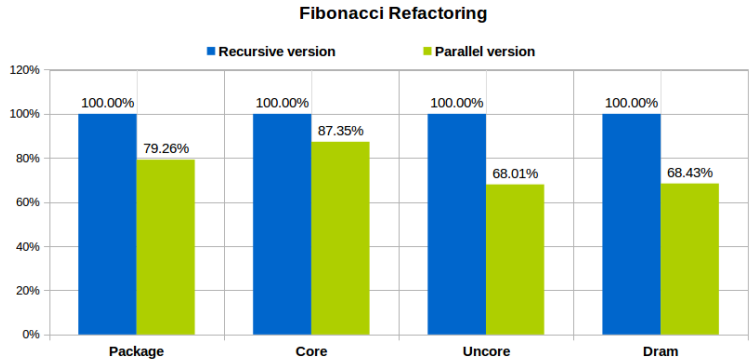
As shown in Figure 5.18 `Recursion version` and `Parallel version` were the most efficient algorithms. This test has been created to analyze which refactoring consumes less energy.

- **Case to test. -** The numbers 40 has been evaluated.
- **Results.-**

Table 5.20: Percentage reduction or increase

	Package	Core	Uncore	Dram	SUM
Recursive version	100.00%	100.00%	100.00%	100.00%	100.00%
Parallel version	79.26%	87.35%	68.01%	68.43%	81.56%

Figure 5.22: Percentage of the energy consumption per variables



- **Analysis**.-The results on the comparison between both implementations are presented in Figure 5.22. `Parallel version` is more efficient than the `Recursive version` even using `Higher order Function`. As shown in Table 5.20 in more efficient with 20% of decrease. The amount in Joules is shown in Table 5.14.

5.2.2 Karatsuba Refactoring

The section 5.1.3 have the description of Karatsuba. To this test the Karatsuba base in `Binary` has been refactored seventeen times. Those refactoring are described in the article "Free the conqueror! refactoring divide-and-conquer functions" by Tamás Kozsik, Melinda Tóth, and István Bozó from Eötvös Loránd University [30]. All the refactorings are measured but just some refactoring have been taken into account.

Source code

Like all the previous tests is important analyze the different algorithms that have been implemented, analyzing the codes. To this test `Karatsuba refactoring` the codes are attached in the Appendix D.

The following are the functions:

- **Karatsuba 1.-** Function Clauses to/from Case Clauses.
- **Karatsuba 2.-** Group Case Branches. Partition the branches of a case-expression into two groups.
- **Karatsuba 3.-** Eliminate Single Branch. When a case-expression contains only a single branch, it can be simplified to an optional match-expression followed by the branch body.
- **Karatsuba 4.-** Extract functions, is_base and solve.
- **Karatsuba 5.-** Eliminate functions, IsBase

- **Karatsuba 6.-** Bindings To List.

- **Karatsuba 7.-** Introduce tuple.

- **Karatsuba 8.-** Introduce Higher order function. The function `lists:map/2` was introduced.

- **Karatsuba 9.-** Introducing variables. The variables SubProblems and SubSolutions was introduced.

- **Karatsuba 10.-** Introducing variables. The variables Old1 and Old2 was introduced.

- **Karatsuba 11.-** Extract functions, divide and combine.

- **Karatsuba 12.-** Generalize function call on is_base.

- **Karatsuba 13.-** Tuple function arguments.

- **Karatsuba 14.-** Introducing tuples against to parameters.

- **Karatsuba 15.-** Extract function binding.

- **Karatsuba 16.-** Introducing functions.

- **Karatsuba 17.-** Introducing parallelism.

- **Case to test.-** The numbers `123456789 161 times (*) 123456789 161 times`. has been evaluated.

- **Results.-** Karatsuba 1 based on `Binary` is taken as Base Case such way that each algorithm is compared with the base case.

Table 5.21: Amount of energy consumption

	Package	**Core**	**Uncore**	**Dram**	**Sum(Joules)**
Karatsuba1	87.55	60.32	0.98	5.42	154.26
Karatsuba2	89.70	61.88	1.04	5.53	158.15
Karatsuba3	89.61	62.06	0.97	5.49	158.13
Karatsuba4	91.40	63.12	1.06	5.64	161.22
Karatsuba5	91.17	63.08	1.10	5.60	160.95
Karatsuba6	90.97	63.17	0.99	5.57	160.69
Karatsuba7	90.98	63.02	1.01	5.58	160.60
Karatsuba8	91.17	63.08	1.10	5.60	160.95
Karatsuba9	94.49	65.54	1.00	5.78	166.80
Karatsuba10	95.52	66.24	1.09	5.85	168.70
Karatsuba11	96.16	66.77	1.14	5.87	169.95
Karatsuba12	96.29	66.49	1.04	5.88	169.70
Karatsuba13	97.62	67.62	1.10	6.00	172.35
Karatsuba14	96.80	67.15	1.09	5.93	170.98
Karatsuba15	97.46	67.73	1.02	5.96	172.17
Karatsuba16	96.87	66.96	1.13	5.95	170.92
Karatsuba17	47.94	35.90	0.38	2.50	86.72

Table 5.22: Recursive version vs All algorithms - Variables

	Package	Core	Uncore	Dram	SUM
Karatsuba 1 (Base Case)	100.00%	100.00%	100.00%	100.00%	100.00%
Karatsuba 2	102.45%	102.60%	106.12%	102.12%	102.52%
Karatsuba 3	102.35%	102.90%	99.01%	101.30%	102.51%
Karatsuba 4	104.40%	104.65%	108.71%	104.02%	104.51%
Karatsuba 5	104.13%	104.59%	113.11%	103.40%	104.34%
Karatsuba 6	103.90%	104.74%	101.01%	102.77%	104.17%
Karatsuba 7	103.91%	104.49%	103.73%	103.05%	104.11%
Karatsuba 8	110.10%	110.57%	113.60%	109.45%	104.34%
Karatsuba 9	107.92%	108.65%	102.09%	106.63%	108.13%
Karatsuba 10	109.10%	109.83%	111.83%	108.01%	109.36%
Karatsuba 11	109.83%	110.70%	116.85%	108.42%	110.17%
Karatsuba 12	109.98%	110.23%	106.87%	108.56%	110.01%
Karatsuba 13	111.50%	112.11%	113.10%	110.70%	111.72%
Karatsuba 14	110.57%	111.34%	112.13%	109.48%	110.84%
Karatsuba 15	111.32%	112.29%	104.72%	110.01%	111.61%
Karatsuba 16	110.64%	111.02%	116.06%	109.82%	110.80%
Karatsuba 17	54.75%	59.52%	38.74%	46.23%	56.22%

Figure 5.23: Recursive version vs all other refactoring - Variables

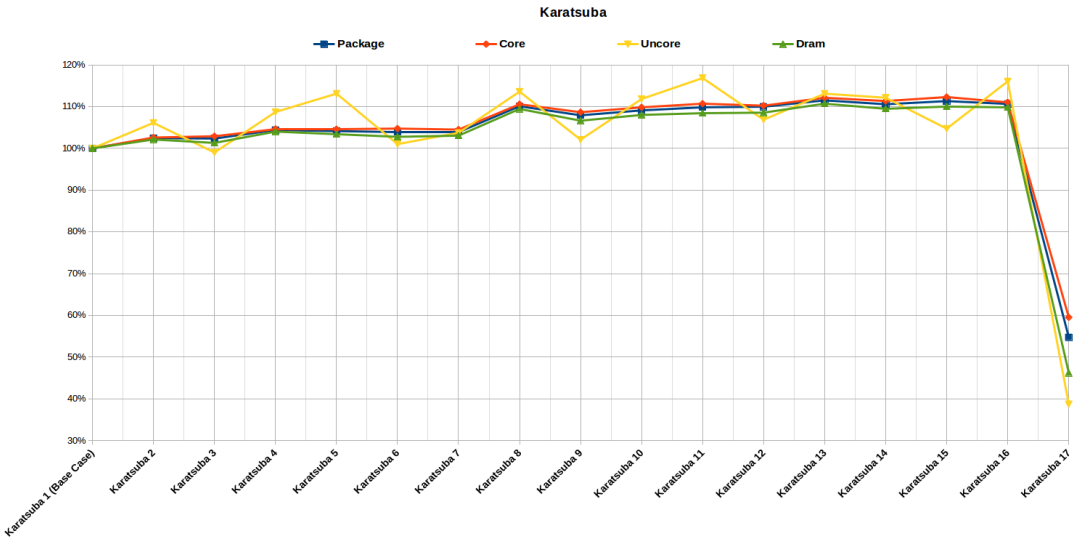
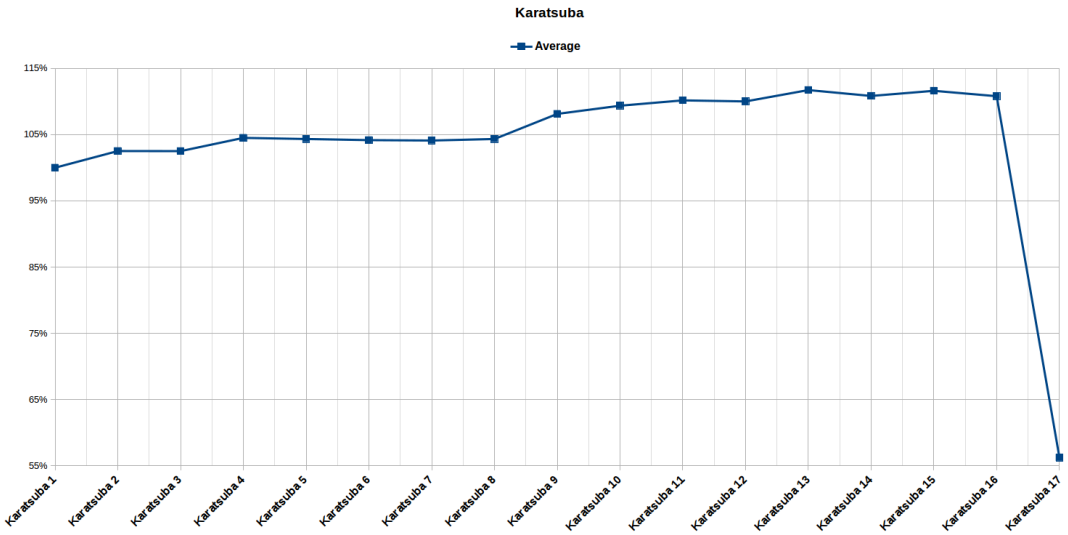


Figure 5.24: Recursive version vs all other refactoring - Average



- **Analysis**.- The amount of refactoring hinder the right analysis of them. Just the relevent cases will be analyzed. The following cases have been analyzed:

  1. Introducing tuples in functions.- Karatsuba 12 vs Karatsuba 13.
  2. Introducing Variables.- Karatsuba 1 vs Karatsuba 2 and and Karatsuba 8 vs Karatsuba 9.
  3. Higher order Function.- Karatsuba 7 vs Karatsuba 8.
  4. Parallel version.- Karatsuba 8 vs Karatsuba 17.

Introducing tuples as parameter

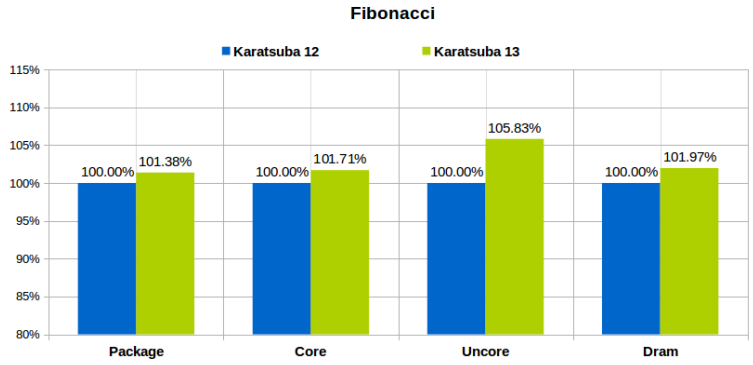
As shown in Figure 5.24 Karatsuba 13 is the refactoring that consumes more energy. This refactoring introduce tuples against variables in the functions. This test has been created to analyze this case.

- **Case to test.-** The numbers `123456789 161 times (*) 123456789 161 times` has been evaluated.
- **Result.-**

Table 5.23: Percentage reduction or increase

	Package	Core	Uncore	Dram	Sum
Karatsuba 12	100.00%	100.00%	100.00%	100.00%	100.00%
Karatsuba 13	101.38%	101.71%	105.83%	101.97%	101.56%

Figure 5.25: Percentage of the energy consumption per variables



- **Analysis**.- The results on the comparison between both implementations are presented in Figure 5.25. The increment of energy with `Introduction of Tuples` is not a considerable amount. The variable `UNCORE` is the maximum increment with almost 6% as shown in Table 5.23. The amount in Joules is shown in Table 5.21.

Introducing Variables

`Introducing Variables` has been analyzed in all the previous tests and is one of the more common refactoring that has been implemented. This test has been created to determine if the `Introduction of variables` affects the energy consumption.

- **Case to test.-** The numbers `123456789 161 times (*) 123456789 161 times` has been evaluated.
- **Result.-**

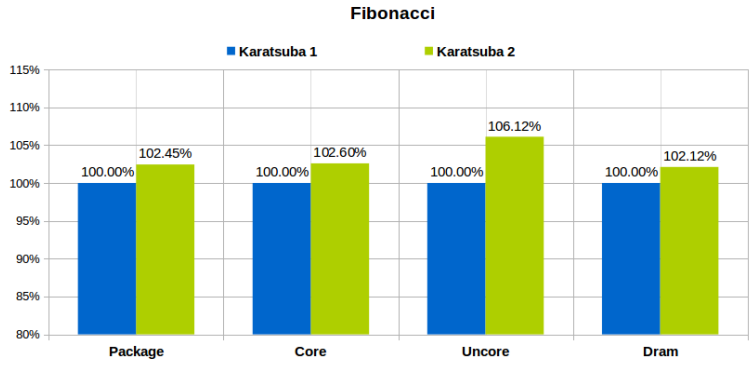
Table 5.24: Percentage reduction or increase - Karatsuba 1 vs Karatsuba 2

	Package	**Core**	**Uncore**	**Dram**	**Sum**
Karatsuba 1	100.00%	100.00%	100.00%	100.00%	100.00%
Karatsuba 2	102.45%	102.60%	106.12%	102.12%	102.52%

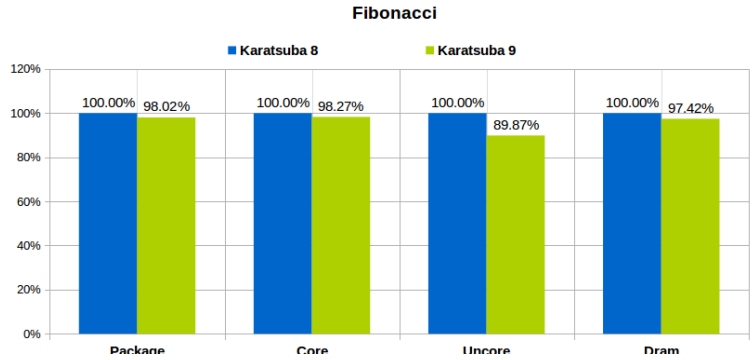
Table 5.25: Percentage reduction or increase - Karatsuba 8 vs Karatsuba 9

	Package	**Core**	**Uncore**	**Dram**	**Sum**
Karatsuba 8	100.00%	100.00%	100.00%	100.00%	100.00%
Karatsuba 9	98.02%	98.27%	89.87%	97.42%	98.05%

Figure 5.26: Percentage of the energy consumption per variables



(a) Introducing variables



(b) Introducing variables

- **Analysis**.- The results on the comparison between both implementations are presented in Figure 5.26. Like the other tests the `Introduction of Variables` is not relevant, in one case is just almost 3% of increment and the other case decrease in almost 2%. The amount in Joules is shown in Table 5.21.

Binary vs Higher order Function

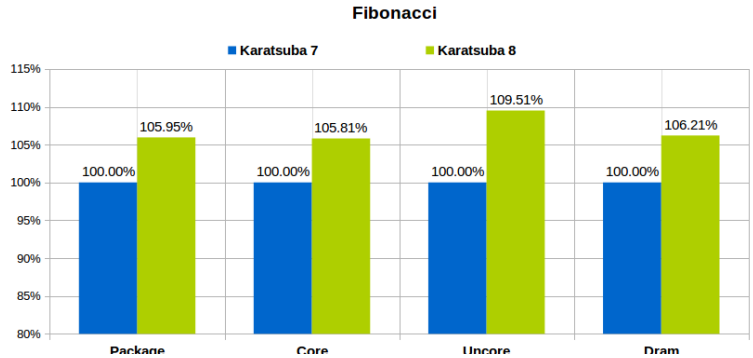
Karatsuba 8 started to apply `lists:map` this means `Higher order Function`. This test has been created to analyze the consumption with `Higher order Function`.

- **Case to test.-** The numbers `123456789 161 times (*) 123456789 161 times` has been evaluated.
- **Result.-**

Table 5.26: Percentage reduction or increase

	Package	Core	Uncore	Dram	Sum
Karatsuba 7	100.00%	100.00%	100.00%	100.00%	100.00%
Karatsuba 8	105.95%	105.81%	109.51%	106.21%	105.93%

Figure 5.27: Percentage of the energy consumption per variables



- **Analysis**.- The results on the comparison between both implementations are presented in Figure 5.27. This case has a different behavior compared with the other cases because there is an increase, but is just the 5.93%, not like the other tests where the increment was greater. The amount in Joules is shown in Table 5.21.

Parallel version vs Higher order Function

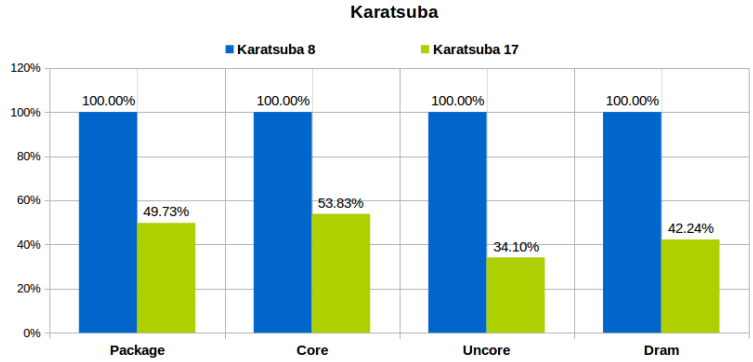
The Karatsuba 17 is applying the `Parallel version`. Karatsuba 8 started to apply `lists:map` this means `Higher order Function`. Based on Karatsuba 8 and Karatsuba 17 this test has been created to analyze which refactoring consume less energy.

- **Case to test.-** The numbers `123456789 161 times (*) 123456789 161 times` has been evaluated.
- **Result.-**

Table 5.27: Percentage reduction or increase

	Package	Core	Uncore	Dram	Sum
Karatsuba 8	100.00%	100.00%	100.00%	100.00%	100.00%
Karatsuba 17	49.73%	53.83%	34.10%	42.24%	50.98%

Figure 5.28: Percentage of the energy consumption per variables



- **Analysis**.- The results on the comparison between both implementations are presented in Figure 5.28. `Parallel version` consume almost the half of energy, in percentage 50.98% of the energy that `Higher order function` consume. The amount in Joules is shown in Table 5.21.

5.3 Discussion

Before to raise the principal conclusions about the energy consumption in the Erlang functions, is important remark the analysis that was made for each experiment. Below are detailed per each test the principal conclusions:

- **First approach.-** Different algorithms
  - **Sum List**
    1. Introduce `Tail recursive function`.- It was the most efficient algorithm.
    2. Introduce `Higher order functions`.- It was the worst algorithms. Introducing `lists:foldl/3`.
    3. `Introduction of variables`.- It does not affect in the energy consumption.
  - **Fibonacci different algorithms**
    1. Reduce `Data structure`.- It reduces the energy consumption.
    2. `Lists`.- It was the worst algorithm implemented.
  - **Karatsuba Binary vs Lists**
    1. Introduce `Simple Lists`.- It was more efficient than Binary.
- **Second approach.-** Different refactorings
  - **Fibonacci Refactoring**
    1. Introduce `Higher order functions`.- It was the less efficient. Introducing `lists:map/2`.
    2. Introduce `Parallelism`.- It was the most efficient.

3. `Introduction of Variables`.- It does not affect in the energy consumption.

– Karatsuba Refactoring
1. `Introduction of Tuples`.- It was the less efficient refactoring, but it was small amount.
2. `Introduction of Variables`.-It does not affect in the energy consumption.
3. Introduce `Higher order Function`.- It had a different behavior comparing with the others functions the increment was small amount. Introducing `lists:map/2`.
4. Introduce `Parallelism`.- It reduces the amount of energy in a big amount.

5.3.1 Effect of Introducing Variables

The `Introduction of Variables` does not affect the energy consumption, due to the fact in some cases increases, others decrease and the amount is not relevant. The following Table is summarized of the tests.

Table 5.28: Analysis of the Introduction of Variables

Introducing Variables in	**Increase**	**Decrease**
Sum List - Tail recursive version		1.43%
Fibonacci Refactoring - Recursive version		11.03%
Fibonacci Refactoring - Higher order Function	1.23%	
Karatsuba Refactoring	2.66%	

As shown in Table 5.28 the `Introduction of Variables` does not affect the energy consumption. In the test `Sum List` the introduction of Variables just reduced 1.43%, in the test `Fibonacci Refactoring` with the `Recursive function` the reduction was of 11.03%. Opposite case in the `Fibonacci Refactoring` but with `Higher order Functions` the energy consumption increase in 1.23%, the same way in `Karatsuba Refactoring` increase in 2.66%. The `Introduction of Variables` has not the same behavior in all the tests, is not possible determinate it is has a negative or positive effect in the energy consumption.

5.3.2 Effect of Using Higher order Functions

`Higher order Function` had the biggest amount of energy consumption. The following Table is summarized the results.

Table 5.29: Analysis of Higher order Functions

Higher order Function VS	**Increase**	**Decrease**
Primitive recursive (Sum List)	11.72%	
Tail recursive (Sum List)	38.21%	
Recursive Function (Fibonacci Refactoring)	307.95%	
Parallel Function (Fibonacci Refactoring)	400.20%	
Recursive Function (Karatsuba)	2.66%	

In Table 5.29 show that `Higher order functions` increment in the energy consumption in all tests. In the test `Sum List` the increment was of 11.72% comparing with the `Primitive recursive function` and comparing with `Tail recursive function` the increment is 38.21%. In the test `Fibonacci Refactoring` the increment was of 307.95% comparing the `Recursive version` and comparing with `Parallel version` was of 400.20%. Finally in `Karatsuba` the behavior was the same but the increment was not so big like the previous tests. The increase was just of 2.66%.

5.3.3 Effect of Improving the Data structure

The reduction of Data Structure reduces the energy consumption like was tested in the section 5.1.2 with Fibonacci series. The following Table summarizes the test.

Table 5.30: Analysis of the Minimal Data structure

	Increase	**Decrease**
Array style vs Minimal Data structure		79.72%
Binary vs List		96.33%

It is interesting to observe that reduction of Data structure reduce the energy consumption in functions, in the case of `Sum Lists` reaching up to 79.72% reduction and in `Karatsuba` reaching up 96.33% as shown in Table 5.30. Both cases are using simples numbers to evaluate.

5.3.4 Effect of Parallel functions

In Fibonacci Refactoring was a very good example to show how the different transformations can influence in the energy consumption. All the transformations were made one after the other, this means that the last transformation has all the previous ones. In such way that the last transformation the `Parallel version` using `Higher order Functions` was the most efficient refactoring. The following Table has the percentages of the difference.

Table 5.31: Recursive version vs Parallel function

	Increase	**Decrease**
Parallel Fibonacci vs Recursive version		18.44%
Parallel Fibonacci vs HOF		420%
Parallel Karatsuba vs HOF		49.02%

As shown in Table 5.31 it is interesting to observe the reduction oof energy consumption when `Parallel functions` is used. In the case of `Fibonacci refactoring` the reduction of 18.44% against of `Recursive function` and 420% against to `Higher order Function`. The same way in `Karatsuba` the reduction of 49% compared with `Higher order Function`.

5.3.5 Effect of type of Recursive Functions

The test `Sum List` tested the `Primitive recursive function` and the `Tail recursive function`. Both are recursive functions and showed that `Tail recursive` is more efficient than `Primitive recursive`. The following Table shown the percentage of reduction.

Table 5.32: Primitive recursive vs Tail recursive

Primitive recursive VS	**Increase**	**Decrease**
Tail recursive version		66.97%

As shown in Table 5.32 the `Tail recursive functions` is able to save until the 66.97% of energy than `Primitive recursive function`.

Chapter 6

Related work

Try to reduce the energy consumption is not a new topic. Exist different levels where we could reduce the energy consumption. For example, operating system, hardware or software. In the case of hardware, improving machine code [31] or optimizing hardware components [32]. Marion Gottschalk and his colleagues in their paper [33] they were focused on the optimization of energy consumption in mobile devices. They propose the detection and remove energy-wasteful code using software re-engineering services, like code analysis and restructuring. The altering source code is viewed as perfective maintenance, but also could help in lowering energy consumption of applications finding energy wasting patterns, those are called energy code smells.

Like a prerequisite removing the code smell, is necessary the identification a cataloging of energy code smells. Exist different classifications of code smells, but the importance of classification the energy code smells are cross-platform as the underlying concepts and mechanisms stay the same on every platform. According to Marion Gottschalk and his team used the next classification:

- Loop Bug,
- Dead code,
- In-line method,
- Moving too much data,
- Immortality Bug,
- Redundant storage of data,
- Using expensive resources.

Once the code smells was classified then they removed the code smells by using refactoring, in order that all of the re-engineers could worked in every platform. The paper show that the re-engineer of the code helps to reduce energy. In the case of Gottschalk the re-engineer was done manually. But, how the code smells were classified before re-engineer, it will help to create

tools in order that the detection and the restructuring are automatic. This technique also can be applied in different on servers and desktops PCs.

In 2013 according with Anne E. Trefethen [34] the energy consumption has become a major concern. But in the beginner the developers have focused on the energy consumption physically at the operational level, but she improves a change this mentality, she was interested in energy-aware software, especially with multi-threading. She studied the NAS Benchmark [35] suite for its energy and runtime performance. This one is purely parallel and includes I/O and compute-intensive applications.

For each benchmark she performed multiple runs. Each run covered different frequencies, thread counts and compilers. Consequently, she could conclude that:

1. There is a clear interaction between runtime and energy, but this one could be affected by the compute environment and algorithmic approach.
2. The size, locality and thread-count all affect the energy consumption.
3. Compilers can have an important impact on the energy consumption of multithreaded applications. By the same token, the software transformations have a key role in runtime performance and energy consumption.
4. The threading improves performance, but the energy consumption maybe not. Because the energy consumption depends on the degree of parallelism and the architecture.
5. The hyper-threading clearly improves the energy efficiency in highly computational applications. And finally,
6. Identifying the interaction between multiple parameters such as frequency, threading and compilers, is possible guarantee substantial energy savings along with considerable performance gain.

She shows in her paper the complexity of finding the optimal energy and multi-threading, but she proves that the numbers of threads could have significant impact on energy consumption for the same level of performance. In order to measure the energy consumption of software, she placed a sensor between the power source and the system, also used PowerPack [36].

The principal aim of refactoring Code benefits are the understandability, maintainability and extensibility and now the automated support for refactoring is common in IDEs. However, when the developers currently do not understand how the choices and tradeoffs they make on a daily basis impact the energy consumption of their software. Even they are focused on quality attributes such as correctness, performance, reliability, and maintainability, no in the energy efficiency [37].

Cagri Sahin with his collages [37] demonstrated that the energy consumption is affected by the scarcity of information about that, for that the developers are not focused on the energy efficiency. They were focused on how different refactorings alter the overall energy consumption of an application, and how can such effects be characterized, predicted, and presented to the software developer for decision making.

In order to show that, they used built-in refactoring tools of Eclipse, to create 197 refactored versions of 9 applications. They executed each refactored version on two different platforms, several times and creating a statistical analysis of the observed impacts of applying refactorings and switching execution platforms on energy usage. Finally, they analyzed over 350 gigabytes of power profiling information collected from 10,300 executions. In them investigation, they used the configuration that eclipse provide like:

- Convert Local Variable to Field,
- Extract Local Variable,
- Extract Method,
- Introduce Indirection,
- Inline Method,
- Introduce Parameter Object.

After of test of the executions they achieved that all the refactorings they considered can statistically significantly impact the energy usage of an application. Likewise, those have the potential to increase and decrease energy usage, with the exception of Extract Local Variable which we only observed to decrease energy usage. Also, they proved that the refactorings is not consistent, this could change depending on applications or platforms. Unfortunately, they notice that the execution time and dynamic execution counts are unlikely to accurately predict the energy impacts of applying a refactoring.

According with Simon Thompson [38] about Refactoring Functional Programming, some refactoring are paradigm-independent:

- some move a definition from one module to another
- change the scope (i.e. Visibility) of a definition.

Instead other have a functional value:

- Replace pattern matching over an algebraic data type with the operations of an abstract datatype
- Generalize a function working over a single type in a polymorphic function applied to a whole class of types
- Replace a function by a constructor.

In addition, he mentioned that the pure functional languages, links to theory tend to be stronger and reasoning about programs, less complicated than in imperative languages. This means that should be possible to import refactorings from the theory of functional languages. Even the differences in programming style and language features like the wide-spread use of algebraic data types,

pattern-matching, and higher-order functions in functional languages, should become apparent in the catalogues of refactorings. At a larger scale, language features supporting programming in the large differ completely, in that functional programming is inherently compositional, objects and object classes are absent, and type theory and logic exert a strong influence on language design. Type systems and type classes will have to be taken into account, as well as patterns for modular programming that build on general abstractions against of specific language features.

Unfortunately exist few researches how the refactoring in functional language could improve the energy consumption. One of those is Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language by Luís Gabriel Lima and his collages [39]. They analyzed the energy behavior of programs written in a lazy purely functional language, like Haskell. They handle two empirical studies to analyze the energy efficiency: strictness and concurrency. Their experimental space exploration comprises more than 2000 configurations and 20000 executions.

They wanted to analyze, to what extent using different data structure implementations or concurrent programming constructs through refactoring, could they save energy. They were raised two empirical studies or scenarios. The first one was analyzed the performance and energy consumption of benchmark operations over fifteen implementations of three different types of data structures. In the second one, they assessed three different thread management constructs and three primitives for data sharing using nine benchmarks and multiple experimental configurations.

In order of better understanding by developers, they used two tools for performance analysis to make them energy-aware, Criterion benchmarking library and the profiler that comes with the Glasgow Haskell Compiler. The experiments that they did were conducted on a machine with 2x10-core Intel Xeon E5-2660 v2 processors (Ivy Bridge microarchitecture) and 256GB of DDR3 1600MHz memory. This machine runs the Ubuntu Server 14.04.3 LTS (kernel 3.19.0-25) OS. The compiler was GHC 7.10.2, using Edison 1.3 (Section IV-A), and a modified Criterion (Section III-B) library. Also, all experiments were performed with no other load on the OS.

For the first analyze they were focused in the strictness for that they used a library of purely functional data structures called *Edison]*. Edison provides different functional data structures to implement three types of abstractions: Sequences, Collections and Associative Collections. They got the following results:

- Sequences.-The observed proportions across all operations and implementations differ at most in 1.9%, for the add operation.
- Set.- For all operations the differences between the proportions of either time or energy consumption are always lower than 1.49%.
- Heaps.- The proportions of runtime and energy consumption differ in at most 2.16% for any operation in any implementation.
- Associative Collections..- The proportion of consumption energy was (marginally, by 1%) higher than the proportion of execution time only for the add operation.

This work was focused on the Haskell programming language. Haskell is a lazy programming language, so perhaps the results do not apply to other functional programming languages.

For the second analyze focus in concurrency they compared concurrent programming constructors. Factors such as operating system scheduling policies and processor and interconnect layouts can clearly impact the results. Nevertheless, after run all of the tests they got the following conclusions:

- Faster is not always greener.
- There is no overall winner.
- Choosing more capabilities than available CPUs is harmful.

Finally they realize that small changes can make a big difference in terms of energy consumption. Like they notice in one benchmarks, with a specific configuration, choosing one data sharing primitive over another can yield 60% energy savings. In another benchmark, the latter primitive can yield up to 30% energy savings over the former. Also, they concluded that the relationship between energy consumption and performance is not always clear. In sequential benchmarks, high performance is an accurate proxy for low energy consumption. Even so, for one of our concurrent benchmarks, the variants with the best performance also exhibited the worst energy consumption.

The refactored should use carefully if the target is reducing energy consumption, because some techniques can consume more power than original codes. Park and his collages in the paper titled "Investigation for Software Power Consumption of Code Refactoring Techniques" [40], they estimated the power consumption for source codes written in C++ programming language. To them the energy efficiency means the degree of how less power can consume to provide the same service or functionality. They estimated the power consumption for every refactoring technique by M. Fowler, with the original code and with the refactor code, and next compared their power consumptions with each other.

They took the sixty three techniques by M. Fowler, and then grouped by the following catalogs:

- Power consumptions for "Composing Method" techniques.
- Power consumptions for "Moving Features Between Objects" techniques.
- Power consumptions for "Organizing Data" techniques.
- Power consumptions for "Simplifying Conditional Expressions" techniques.
- Power consumptions for "Making Methods Calls Simpler" techniques.
- Power consumptions for "Dealing with Generalization" techniques.

After the tests they realize that 26 out of the 63 techniques reveal the decrement of power consumption, and the 7 techniques have no changes in their consumption. Others consume more power than those of original code.

The refactoring technicals use different methods like:

- Extraction or composition of module/variable.- Extraction increases the power consumption, because of the addition of the new module to the original code. But the module composition method decreases the power consumption because of the reduced interactions.
- Movement of module/variable.- Decreases the power consumption due to removal of unnecessary relationships inherent in original code.
- Replacement of module/variable.- The power consumption varies depending on the changing forms.
- Introduction or removal of module/variable.- It increases the power consumption when something is added and decrease when something is deleted.

But they also alert that the all technicals work like previously mentioned. In order to save energy also they mentioned, substitute algorithm Technique and Pull Up Method vs. Pull Up Constructor Body. However, if a software engineer wants to select the other technique which was excluded in both tables, it requires any other processing to compensate the loss of energy efficiency. We propose the next two strategies to the compensation.

- Strategy 1: substitute it for a possible technique which has better energy efficiency.
- Strategy 2: redesign the refactoring process to improve energy efficiency.

Finally, they estimated and analyzed the power consumption of the 63 refactoring techniques of Martin Fowler, and recommended 33 techniques among them as energy-efficient refactoring ones. But how all the analyzes of energy consumptions some technicals could work in some cases and in others no or may exists some other techniques to be applied in their practice.

Another problem of Refactoring to reduce the energy consumption, is the insufficient information and tools to help with the measures the different characteristic of the computer energy, like CPU-intensive, memory-intensive,etc [39]. For example, there are libraries like RAPL, Criterion or GHC profile [39].

- RAPL [41] .- Is an interface provided by modern Intel processors to allow setting custom power limits to the processor packages. Through model-specific register (MSR) is a possible access energy and power readings. RAPL estimate the energy consumption based on various hardware performance counters, temperature, leakage models and I/O models.
- Criterion [42].-Is a microbenchmarking library that is used to measure the performance of Haskell code. Criterion is able to measure CPU time, CPU cycles, memory allocation and garbage collection. Also is able to measure events with duration in the order of picoseconds.
- GHC profiler.- Is capable of measuring time and space usage. Developers can enable profiling by compiling a program with the -prof flag.

Like Luis Gabriel Lima [39] said in his paper, to this thesis one of the principal problems was found a tool that is available to measure the energy consumption of Erlangs functions. However, one of the tools was modified and it was available to measure the functions. In the same way differents, refactorings were applied. One of the techniques applied was one taken of M.Fowleer "Simplifying conditional expressions" in the function `Sum Array - Minimal Data Structure`, were the reduction of Data structure reduced the amount of energy consumption. Likewise, the refactoring "Replacement of module/variable" in the case `Introduction of Variables` in many cases was applied, and show that does not affect in the energy consumption.

Chapter 7

Conclusions

Green Computing have gained importance in the last years. Either in physical or logical fields the Green Computing is working. Physically, companies started to use different alternatives that are more friendly to the environment for example, to get the electricity use hydroelectric, etc. Logically, the developers are creating efficient functions and the same time those consume the minimal amount of energy that is possible.

Erlang is a functional language program that is growing fast. Big companies started to use this language due to the fact that has very good qualities in the efficiency of functions. This is a new topic in Erlang, therefore, there is not a previous research about that. Even at the time that the thesis was made, there were not an available tool in order to measure the amount of energy that the Erlang functions consume. It was necessary the creation of a new software to measure the amount of energy consumption in Erlang functions. As a result of the research and testing, joining Read_RAPL software and Erlangs ports were the best option.

Different functions were created to analyze different scenarios. The functions were refactored in order to see if the different transformations influence on the energy consumption. All the tests were tested with the same methodology in order to not get wrong results. The following functions were measured with different algorithms and different refactorings:

- Sum List.- Four different algorithms.
- Fibonacci.- Four different algorithms and seven refactorings.
- Karatsuba.- Two different algorithms and seventeen refactorings.

After executing all the tests this thesis I found the following patterns:

- **Introducing variables**.- The section 5.3.1, showed that in some case the energy consumption increase and in other cases decrease. The percentage of increase or decrease is not enough to be considered. The Introduction of variables does not have the same behavior, is

not possible to considerer that the Introduction of variables affects positively or negatively on the energy consumption.

- **Using Higher order functions.-** The section 5.3.2 show that to all the tests, the Introduction of Higher order Function affected negatively with the energy consumption. In some tests, the amount of energy consumption is extremely huge. However, the case of Karatsuba had a different behavior, despite there is an increment the amount is not big. When the target is save energy the Higher order functions has to be analyzed carefully.
- **Using Parallel functions.-** The section 5.3.4 show that to all the tests, use Parallel version reduced the amount of energy consumption in a big amount. Even when those functions were used Higher order functions that always was inefficient. Definitely, one way in order to save energy is used Parallel functions.

It is also possible to mention the following recommendations:

- The section 5.3.5 present that Tail recursive functions consume less energy than Primitive recursive functions. Try to avoid Primitive recursive function in order to save energy.
- Reduce the Data structure helps in the reduction of the energy consumption in Erlang functions, so is recommended to develop functions with the less possible Data structure.

It is important to consider is that the patterns and the recommendations must be analyzed depending on the environment of the software, keeping in mind all the characteristics of the computer or server. As it is known the energy consumption depends on many facts, so it is not possible to mention that applying those patterns or recommendations has to reduce the energy consumption. Instead is a guide for the developers when they are developing softwares and one of the targets is to create functions that reduce the energy consumption in Erlang.

As mentioned in the beginning "Green Computing in Erlang" is a new topic. This means that the topic has many fields to be discovered and this Thesis is just a small beginning. Will be interesing continue testing more refactored Erlang functions. Also like a future work, is necessary create a tool that detects automatically the energy consumption, to this thesis the measurement was manually. New automatic software will be really useful to most accurate measurements and save time. Finally, keep in mind that we have just one planet, one Earth and that any small change or minor help will be useful for our planet.

Bibliography

[1] NASA. 2016 Climate Trends Continue to Break Records. https://www.nasa.gov/feature/goddard/2016/climate-trends-continue-to-break-records/. [Accessed: 2017.04.11].

[2] PHYS. Natural disasters since 1900—over 8 million deaths and 7 trillion US dollars damage. https://phys.org/news/2016-04-natural-disasters-1900over-million-deaths.html. [Accessed: 2017.04.13].

[3] Techopedia. Green Computing. https://www.techopedia.com/definition/14753/green-computing. [Accessed: 2017.04.11].

[4] The future of things. Green Computing. http://thefutureofthings.com/3083-green-computing/. [Accessed: 2017.04.13].

[5] Andy Hooper. Green computing. *Communication of the ACM*, 51(10):11–13, 2008.

[6] Computing now. Green it and green software. https://www.computer.org/web/computingnow/archive/october2014. [Accessed: 2017.05.22].

[7] Fred Hébert. *Learn You Some Erlang for Great Good!: A Beginner's Guide*. No Starch Press, 2013.

[8] Erlang. Getting Start. https://www.erlang.org/. [Accessed: 2017.04.11].

[9] Erlang. Introduction. http://erlang.org/doc/system_architecture_intro/sys_arch_intro.html#id58791. [Accessed: 2017.04.11].

[10] Wikipedia. Erlang (programming language). https://en.wikipedia.org/wiki/Erlang_(programming_language). [Accessed: 2017.04.11].

[11] Wikipedia. Functional programming. https://en.wikipedia.org/wiki/Functional_programming. [Accessed: 2017.04.11].

[12] Colin Runciman, Amanda Clare, and Rob Harkness. Laboratory automation in a functional programming language. *Journal of laboratory automation*, page 2211068214543373, 2014.

[13] Martin Odersky and Tiark Rompf. Unifying functional and object-oriented programming with scala. *Communications of the ACM*, 57(4):76–86, 2014.

[14] DZone. Functional programming. https://dzone.com/articles/concept-functional-programing. [Accessed: 2017.04.12].

[15] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[16] Refactoring. Refactoring. `https://refactoring.com/`. [Accessed: 2017.04.12].

[17] RefactorErl. Eötvös loránd university. `http://plc.inf.elte.hu/erlang/index.html`. [Accessed: 2017.05.22].

[18] INTEL. RUNNING AVERAGE POWER LIMIT – RAPL. `https://01.org/blogs/2014/running-average-power-limit-\T1\textendash-rapl/`. [Accessed: 2017.04.12].

[19] ubuntu. Turbostat. `http://manpages.ubuntu.com/manpages/xenial/man8/turbostat.8.html`. [Accessed: 2017.05.07].

[20] University of Tennessee. PAPI . `http://icl.utk.edu/papi/`. [Accessed: 2017.05.07].

[21] drexel. PAPI on ubuntu. `https://www.cs.drexel.edu/~tc365/papi361u804.html`. [Accessed: 2017.05.07].

[22] Wiki Kernel. Linux kernel profiling with perf. `https://perf.wiki.kernel.org/index.php/Tutorial`. [Accessed: 2017.05.07].

[23] Linux. Linux thermal daemon monitors and controls temperature in tablets, laptops. `https://www.linux.com/blog/linux-thermal-daemon-monitors-and-controls-temperature-tablets-laptops`. [Accessed: 2017.05.07].

[24] Erlang Factory. Measuring erlang energy consumption, and why this matters. `http://www.erlang-factory.com/sfbay2017/filipe-varjao.html`. [Accessed: 2017.05.14].

[25] kliu20. JRAPL. `http://kliu20.github.io/jRAPL/`. [Accessed: 2017.04.12].

[26] Erlang.org. Ports and port drivers. `http://erlang.org/doc/reference_manual/ports.html`. [Accessed: 2017.05.22].

[27] wikipedia. Summation. `https://en.wikipedia.org/wiki/Summation`. [Accessed: 2017.05.03].

[28] Wikipedia. Fibonacci number. `https://en.wikipedia.org/wiki/Fibonacci_number#CITEREFBeckGeoghegan2010`. [Accessed: 2017.04.24].

[29] wikipedia. Karatsuba algorithm. `https://en.wikipedia.org/wiki/Karatsuba_algorithm`. [Accessed: 2017.05.03].

[30] István Bozó Tamás Kozsik, Melinda Tóth. Free the conqueror! refactoring divide-and-conquer functions. *To appear in Future Generation Computer Systems 2017*, page 26, 2017.

[31] Kaushik Roy and Mark C Johnson. Software design for low power. In *Low power design in deep submicron electronics*, pages 433–460. Springer, 1997.

[32] Hagen Höpfner and Christian Bunse. Energy awareness needs a rethinking in software development. In *ICSOFT (2)*, pages 294–297, 2011.

[33] Marion Gottschalk, Mirco Josefiok, Jan Jelschen, and Andreas Winter. Removing energy code smells with reengineering services. *GI-Jahrestagung*, 208:441–455, 2012.

[34] Anne E Trefethen and Jeyarajan Thiyagalingam. Energy-aware software: Challenges, opportunities and strategies. *Journal of Computational Science*, 4(6):444–449, 2013.

[35] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[36] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li, and Kirk W Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):658–671, 2010.

[37] Cagri Sahin, Lori Pollock, and James Clause. How do code refactorings affect energy usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 36. ACM, 2014.

[38] Simon Thompson. Refactoring functional programs. In *International School on Advanced Functional Programming*, pages 331–357. Springer, 2004.

[39] 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.

[40] Jae Jin Park, Jang-Eui Hong, and Sang-Ho Lee. Investigation for software power consumption of code refactoring techniques. In *SEKE*, pages 717–722, 2014.

[41] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. Rapl: memory power estimation and capping. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pages 189–194. IEEE, 2010.

[42] Bryan O'Sullivan. criterion: Robust, reliable performance measurement and analysis. *Webseite http://hackage. haskell. org/package/criterion, zuletzt abgerufen am*, 31, 2010.

Appendix A

Different algorithms - Fibonacci serie

```
1
2  %--------------------------ARRAY STYLE-------------------------------------
3
4  fib_arr(0) -> 0;
5  fib_arr(1) -> 1;
6  fib_arr(N) ->
7      Begin = 0,
8      End = N + 1,
9      fib_arr(N, End, 2, array:from_list(lists:seq(Begin, End))).
10
11 fib_arr(N, End, I, Fibs) when I == End -> array:get(N, Fibs);
12 fib_arr(N, End, I, Fibs) ->
13     Fib = array:get(I-1, Fibs) + array:get(I-2, Fibs),
14     fib_arr(N, End, I+1, array:set(I, Fib, Fibs)).
15
16
17 %--------------------------LIST STRUCTURE -----------------------------------
18
19 fib_list_naive(0) -> 0;
20 fib_list_naive(1) -> 1;
21 fib_list_naive(N) ->
22     fib_list_naive(N + 2, 3, [0, 1]).
23
24 fib_list_naive(End, I, Fibs) when I == End -> lists:last(Fibs);
25 fib_list_naive(End, I, Fibs) ->
26     Fib = lists:nth(I-1, Fibs) + lists:nth(I-2, Fibs),
27     fib_list_naive(End, I+1, Fibs++[Fib]).
```

```
%---------------------------REVERSE ORDER LISTS---------------------------------

fib_list(0) -> 0;
fib_list(1) -> 1;
fib_list(N) ->
    fib_list(N + 1, [1,0]).

fib_list(End, [H|_]=L) when length(L) == End -> H;
fib_list(End, [A,B|_]=L) ->
    fib_list(End, [A+B|L]).

%---------------------------MINIMAL DATA STRUCTURE-------------------------------
fib_arith(N) when N > 0 -> fib_arith(N, 0, 1).
fib_arith(0, F1, _F2) -> F1;
fib_arith(N, F1,  F2) -> fib_arith(N - 1, F2, F1 + F2).
```

Appendix B

Different algorithms - Fibonacci serie

```
%--------------------------------- BINARY LISTS -----------------------------
karatsuba(A,B) ->
    X = bit_size(A),
    Y = bit_size(B),
    if
        X < Y -> karatsuba_(shifte(A, Y-X), B);
        X > Y -> karatsuba_(A, shifte(B, X-Y));
        true -> karatsuba_(A, B)
    end.

karatsuba_(<<>>, _)->
    <<>>;
karatsuba_(_, <<>>)->
    <<>>;
karatsuba_(Num1, Num2) ->
    case {Num1, Num2} of
        {<<0:1>>, _} -> <<0:(bit_size(Num2))>>;
        {<<1:1>>, _} -> Num2;
        {_, <<0:1>>} -> <<0:(bit_size(Num1))>>;
        {_, <<1:1>>} -> Num1;
        _ ->
            M = max(bit_size(Num1), bit_size(Num2)),
            M2 = M - (M div 2),
            <<Low1:M2/bitstring, High1/bitstring>> = Num1,
            <<Low2:M2/bitstring, High2/bitstring>> = Num2,
            Z0 = karatsuba_(Low1, Low2),
            Z1 = karatsuba_(add(Low1, High1), add(Low2, High2)),
            Z2 = karatsuba_(High1, High2),
            add(add(shift(Z2, M2 * 2), Z0), shift(sub(Z1, add(Z2, Z0)), M2))
    end.
```

```
1
2  shift(B, P)->
3      <<0:P, B/bitstring>>.
4
5  shifte(B, P)->
6      <<B/bitstring, 0:P>>.
7
8  add(A, B)->
9      X = bit_size(A),
10     Y = bit_size(B),
11     if
12         X < Y -> add(shifte(A, Y-X), B, 0);
13         X > Y -> add(A, shifte(B, X-Y), 0);
14         true -> add(A, B, 0)
15     end.
16
17
18 add(<<>>,<<>>,0)->
19     <<>>;
20 add(<<>>,<<>>,1)->
21     <<1:1>>;
22 add(<<X:1,Y/bitstring>>, <<Z:1, V/bitstring>>, C)->
23      R = X + Z + C,
24      if
25          R =< 1 ->
26              End = add(Y, V, 0),
27              <<R:1, End/bitstring>>;
28          R == 2 ->
29              End = add(Y, V, 1),
30              <<0:1, End/bitstring>>;
31          R == 3 ->
32              End = add(Y, V, 1),
33              <<1:1, End/bitstring>>
34      end.
35
36
37 sub(A, B)->
38     AS = bit_size(A),
39     BS = bit_size(B),
40     if
41         AS < BS -> sub(shifte(A, BS-AS), B, 0);
42         AS > BS -> sub(A, shifte(B, AS-BS), 0);
43         true -> sub(A, B, 0)
44     end.
45
46 sub(<<>>,<<>>,_) ->
47     <<>>;
48 sub(<<X:1, Y/bitstring>>, <<Z:1, V/bitstring>>, C) ->
49     R = X - Z - C,
50     if
51         R >= 0 ->
52             End = sub(Y, V, 0),
53             <<R:1, End/bitstring>>;
54         R == -1 ->
55             End = sub(Y, V, 1),
56             <<1:1, End/bitstring>>;
57         R == -2 ->
58             End = sub(Y, V, 1),
59             <<0:1, End/bitstring>>
60     end.
```

```
%----------------------------------- LISTS --------------------------------
big_mult(N1,N2) ->
  S1 = integer_to_list(N1),
  S2 = integer_to_list(N2),
  L1 = lists:reverse(stringToIntList(S1)),
  L2 = lists:reverse(stringToIntList(S2)),
  convertListToInt(lists:reverse(karatsuba_2(L1,L2))).

convertIntToList(N) ->
  stringToIntList(integer_to_list(N)).

stringToIntList(List) ->
  [list_to_integer(X) || X<-[[Y] || Y<-List]].

convertListToInt(List) ->
  L = [integer_to_list(Y) || Y<-List],
  list_to_integer(helperIntListToString(L)).

helperIntListToString([]) ->
  "";

helperIntListToString([H|T]) ->
  H ++ helperIntListToString(T).

addZerosToIntList(NZeros, L) ->
  if NZeros > 0 -> addZerosToIntList(NZeros-1,L ++ [0]);
     true -> L
  end.

opLists(_, [], []) -> [];

opLists(_, _, []) -> [];

opLists(_, [], _) -> [];

opLists(Op, L1, L2) ->
  if
    Op == "+" -> opLAdd(L1, L2);
    Op == "-" -> opLSub(L1, L2);
    Op == "*" -> opLMul(L1, L2)
  end.

calculateN(L1,L2) ->
  max(length(L1), length(L2)) div 2.

getSecondDigit(N) ->
  N rem 10.

opLAdd(L1, []) ->
  L1;

opLAdd([],L2) ->
  L2;
```

```
opLAdd([H1|T1],[H2|T2]) ->
  H = H1 + H2,
  if (H < 10)   -> [H] ++ (opLAdd(T1,T2));
     (H >= 10)  -> if (length(T1) > 0)                    ->
     [getSecondDigit(H)] ++ opLAdd(([hd(T1)+1] ++ tl(T1)),T2);
                  ((length(T1) == 0) and (length(T2) > 0))  ->
                  [getSecondDigit(H)] ++ opLAdd(T1,([hd(T2)+1] ++ tl(T2)));
                  ((length(T1) == 0) and (length(T1) == 0)) ->
                  [getSecondDigit(H)] ++ [1]
                  end
  end.

opLSub(L1,[]) ->
  L1;

opLSub([],L2) ->
  L2;

opLSub([H1|T1],[H2|T2]) ->
  H = H1 - H2,
  if (H >= 0) ->
        [H] ++ (opLSub(T1,T2));
         (H < 0)  ->
                H3 = H1 - H2 + 10,
        if (length(T1) > 0) ->
                [H3] ++ opLSub(([hd(T1)-1] ++ tl(T1)),T2);
                ((length(T1) == 0) and (length(T2) > 0))  ->
                        [H3] ++ opLSub(T1,([hd(T2)-1] ++ tl(T2)));
        ((length(T1) == 0) and (length(T1) == 0)) ->
                [H3]
                end
        end.

opLMul(L,N) ->
  lists:reverse(addZerosToIntList(N,lists:reverse(L))).

karatsuba_2(L1,L2) ->
  if
    (length(L1) < 5) or (length(L2) < 5) ->
     lists:reverse(convertIntToList(convertListToInt(lists:reverse(L1))
     * convertListToInt(lists:reverse(L2))));
    true ->
      Diff = length(L1) - length(L2),
      {L11, L22} =
      if
        (Diff > 0) -> {L1, addZerosToIntList(Diff,L2)};
        (Diff < 0) -> {addZerosToIntList((Diff*-1),L1), L2};
        true       -> {L1, L2}
      end,
      N = calculateN(L11,L22),
      {Low1, High1} = lists:split(N,L11),
      {Low2, High2} = lists:split(N,L22),
      %%{Low1,High1,Low2,High2}
      Z0 = karatsuba_2(Low1,Low2),
      Z1 = karatsuba_2(opLists("+",Low1,High1),opLists("+",Low2,High2)),
      Z2 = karatsuba_2(High1,High2),
      %%{Z0,Z1,Z2}
      (opLists("+",opLists("+",opLists("*",Z2,(2*N)),opLists("*",opLists("-",
      opLists("-",Z1,Z2)),Z0),N)),Z0))

  end.
```

Appendix C

Refactoring - Fibonacci Serie

```
1
2  % ----------------------  RECURSION ----------------------------------
3  fib_1(0) ->    0;
4  fib_1(1) ->    1;
5  fib_1(N) ->
6      fib_1(N-1) + fib_1(N-2).
7
8  % --------------------  INTRODUCING VARIABLES-------------------------
9  % Introducing variables
10 fib_2(1) -> 1;
11 fib_2(0) -> 0;
12 fib_2(N) ->
13     A = fib_2(N - 1),
14     B = fib_2(N - 2),
15     A + B.
16
17 % --------------------  INTRODUCING BINDING LISTS-------------------------
18 %%% Bindings to list
19 fib_3(1) -> 1;
20 fib_3(0) -> 0;
21 fib_3(N) ->
22     [A, B] = [fib_3(N - 1), fib_3(N - 2)],
23     A + B.
24
25 % ----------------- INTRODUCING HIGHER ORDER FUNCTION --------------------
26
27 fib_4(1) -> 1;
28 fib_4(0) -> 0;
29 fib_4(N) ->
30     [A, B] = lists:map(fun fib_4/1, [N - 1, N - 2]),
31     A + B.
```

```
% --------------------  INTRODUCING VARIABLES - HOF------------------------
%%%% Introduce variables

fib_5(1) -> 1;
fib_5(0) -> 0;
fib_5(N) ->
    SubPr = [N - 1, N - 2],
    [A, B] = lists:map(fun fib_5/1, SubPr),
    A + B.

% --------------------  INTRODUCING VARIABLES 2 - HOF------------------------
fib_6(1) -> 1;
fib_6(0) -> 0;
fib_6(N) ->
    SubPr = [N - 1, N - 2],
    SubSols = lists:map(fun fib_6/1, SubPr),
    [A, B] = SubSols,
    A + B.

% ----------------------------  PARALLEL VERSION -------------------------------
fib_7(1) -> 1;
fib_7(0) -> 0;
fib_7(N) ->
    SubPr = [N - 1, N - 2],
    SubSols = pmap(fun fib_p/1, SubPr),
    lists:sum(SubSols).
```

Appendix D

Refactoring - Karatsuba

```
1
2  karatsuba1( Num1 , Num2) ->
3      S1 = bit_size(Num1),
4      S2 = bit_size(Num2),
5      case {Num1, Num2} of
6          {<<0:1>>, _} -> <<0: S2>>;
7          {_, <<0:1>>} -> <<0: S1>>;
8          {<<1:1>>, _} -> Num2;
9          {_, <<1:1>>} -> Num1;
10         _            ->
11             M = max( S1, S2 ),
12             M2 = M - (M div 2),
13             <<Low1 : M2/bitstring, High1/bitstring>> = Num1,
14             <<Low2 : M2/bitstring, High2/bitstring>> = Num2,
15             Z0 = karatsuba1( Low1, Low2),
16             Z1 = karatsuba1( add(Low1,High1), add(Low2,High2)),
17             Z2 = karatsuba1( High1, High2),
18             add( add( shift(Z2, M2*2), Z0 ),
19                  shift( sub(Z1, add(Z2,Z0)), M2 ) )
20     end.
```

```
%%%%% Group case expression branches
karatsuba2( Num1 , Num2) ->
    S1 = bit_size( Num1 ),
    S2 = bit_size( Num2 ),
    IsBase = case {Num1, Num2} of
            {<<0:1>>, _} -> true;
            {_, <<0:1>>} -> true;
            {<<1:1>>, _} -> true;
            {_, <<1:1>>} -> true;
            _ -> false
        end,
    case IsBase of
        true ->
            case {Num1, Num2} of
                {<<0:1>>, _} -> <<0:S2>>;
                {_, <<0:1>>} -> <<0:S1>>;
                {<<1:1>>, _} -> Num2;
                {_, <<1:1>>} -> Num1
            end;
        false ->
            case {Num1, Num2} of
                _ ->
                    M = max(S1, S2),
                    M2 = M - (M div 2),
                    <<Low1:M2/bitstring, High1/bitstring>> = Num1,
                    <<Low2:M2/bitstring, High2/bitstring>> = Num2,
                    Z0 = karatsuba2(Low1, Low2),
                    Z1 = karatsuba2(add(Low1, High1), add(Low2, High2)),
                    Z2 = karatsuba2(High1, High2),
                    add(add(shift(Z2, M2 * 2), Z0),
                        shift(sub(Z1, add(Z2, Z0)), M2))
            end
    end.
```

```
%%%%% Eliminate single branch
karatsuba3( Num1 , Num2) ->
    S1 = bit_size( Num1 ),
    S2 = bit_size( Num2 ),
    IsBase = case {Num1, Num2} of
            {<<0:1>>, _} -> true;
            {_, <<0:1>>} -> true;
            {<<1:1>>, _} -> true;
            {_, <<1:1>>} -> true;
            _ -> false
        end,
    case IsBase of
        true ->
            case {Num1, Num2} of
                {<<0:1>>, _} -> <<0:S2>>;
                {_, <<0:1>>} -> <<0:S1>>;
                {<<1:1>>, _} -> Num2;
                {_, <<1:1>>} -> Num1
            end;
        false ->
            _ = {Num1, Num2},
            M = max(S1, S2) ,
            M2 = M - (M div 2) ,
            <<Low1:M2/bitstring, High1/bitstring>> = Num1,
            <<Low2:M2/bitstring, High2/bitstring>> = Num2,
            Z0 = karatsuba3(Low1, Low2),
            Z1 = karatsuba3(add(Low1, High1), add(Low2, High2)),
            Z2 = karatsuba3(High1, High2),
            add(add(shift(Z2, M2 * 2), Z0),
                shift(sub(Z1, add(Z2, Z0)), M2))
    end.

%%%%% Extract is_base and solve
karatsuba4( Num1 , Num2) ->
    S1 = bit_size( Num1 ),
    S2 = bit_size( Num2 ),
    IsBase = is_base(Num1, Num2),
    case IsBase of
        true ->
            solve(Num1, Num2, S1, S2);
        false ->
            _ = {Num1, Num2},
            M = max(S1, S2) ,
            M2 = M - (M div 2) ,
            <<Low1:M2/bitstring, High1/bitstring>> = Num1,
            <<Low2:M2/bitstring, High2/bitstring>> = Num2,
            Z0 = karatsuba4(Low1, Low2),
            Z1 = karatsuba4(add(Low1, High1), add(Low2, High2)),
            Z2 = karatsuba4(High1, High2),
            add(add(shift(Z2, M2 * 2), Z0),
                shift(sub(Z1, add(Z2, Z0)), M2))
    end.

is_base(Num1, Num2) ->
    case {Num1, Num2} of
        {<<0:1>>, _} -> true;
        {_, <<0:1>>} -> true;
        {<<1:1>>, _} -> true;
        {_, <<1:1>>} -> true;
        _ -> false
    end.
```

```
solve(Num1, Num2, S1, S2) ->
    case {Num1, Num2} of
        {<<0:1>>, _} -> <<0:S2>>;
        {_, <<0:1>>} -> <<0:S1>>;
        {<<1:1>>, _} -> Num2;
        {_, <<1:1>>} -> Num1
    end.

%%%% Eliminate IsBase:
karatsuba5( Num1 , Num2) ->
    S1 = bit_size( Num1 ),
    S2 = bit_size( Num2 ),
    case is_base(Num1, Num2) of
        true ->
            solve(Num1, Num2, S1, S2);
        false ->
            _ = {Num1, Num2},
            M = max(S1, S2) ,
            M2 = M - (M div 2) ,
            <<Low1:M2/bitstring, High1/bitstring>> = Num1,
            <<Low2:M2/bitstring, High2/bitstring>> = Num2,
            Z0 = karatsuba5(Low1, Low2),
            Z1 = karatsuba5(add(Low1, High1), add(Low2, High2)),
            Z2 = karatsuba5(High1, High2),
            add(add(shift(Z2, M2 * 2), Z0),
                shift(sub(Z1, add(Z2, Z0)), M2))
    end.

%%%% Sol1, ..., SolN already introduced so performe Bindings to list
karatsuba6( Num1 , Num2) ->
    S1 = bit_size( Num1 ),
    S2 = bit_size( Num2 ),
    case is_base(Num1, Num2) of
        true ->
            solve(Num1, Num2, S1, S2);
        false ->
            _ = {Num1, Num2},
            M = max(S1, S2) ,
            M2 = M - (M div 2) ,
            <<Low1:M2/bitstring, High1/bitstring>> = Num1,
            <<Low2:M2/bitstring, High2/bitstring>> = Num2,
            [Z0, Z1, Z2] = [karatsuba6(Low1, Low2),
                karatsuba6(add(Low1, High1), add(Low2, High2)),
                karatsuba6(High1, High2)],
            add(add(shift(Z2, M2 * 2), Z0),
                shift(sub(Z1, add(Z2, Z0)), M2))
    end.
```

```
%%%% Introduce tuple
karatsuba7({Num1, Num2}) ->
    S1 = bit_size( Num1 ),
    S2 = bit_size( Num2 ),
    case is_base(Num1, Num2) of
        true ->
            solve(Num1, Num2, S1, S2);
        false ->
            _ = {Num1, Num2},
            M = max(S1, S2) ,
            M2 = M - (M div 2) ,
            <<Low1:M2/bitstring, High1/bitstring>> = Num1,
            <<Low2:M2/bitstring, High2/bitstring>> = Num2,
            [Z0, Z1, Z2] = [karatsuba7({Low1, Low2}),
                karatsuba7({add(Low1, High1), add(Low2, High2)}),
                karatsuba7({High1, High2})],
            add(add(shift(Z2, M2 * 2), Z0),
                shift(sub(Z1, add(Z2, Z0)), M2))
    end.

%%%% Calls to map
karatsuba8({Num1, Num2}) ->
    S1 = bit_size( Num1 ),
    S2 = bit_size( Num2 ),
    case is_base(Num1, Num2) of
        true ->
            solve(Num1, Num2, S1, S2);
        false ->
            _ = {Num1, Num2},
            M = max(S1, S2) ,
            M2 = M - (M div 2) ,
            <<Low1:M2/bitstring, High1/bitstring>> = Num1,
            <<Low2:M2/bitstring, High2/bitstring>> = Num2,
            [Z0, Z1, Z2] = pmap(fun karatsuba8/1,
                    [{Low1, Low2}, {add(Low1, High1), add(Low2, High2)},
                    {High1, High2}]),
            add(add(shift(Z2, M2 * 2), Z0),
                shift(sub(Z1, add(Z2, Z0)), M2))
    end.

%%%% Introduce SubProblems and SubSolutions
karatsuba9({Num1, Num2}) ->
    S1 = bit_size( Num1 ),
    S2 = bit_size( Num2 ),
    case is_base(Num1, Num2) of
        true ->
            solve(Num1, Num2, S1, S2);
        false ->
            _ = {Num1, Num2},
            M = max(S1, S2),
            M2 = M - (M div 2),
            <<Low1:M2/bitstring, High1/bitstring>> = Num1,
            <<Low2:M2/bitstring, High2/bitstring>> = Num2,
            SubProblems = [{Low1, Low2}, {add(Low1, High1), add(Low2, High2)},
                {High1, High2}],
            SubSolutions = lists:map(fun karatsuba9/1, SubProblems),
            [Z0, Z1, Z2] = SubSolutions,
            add(add(shift(Z2, M2 * 2), Z0),
                shift(sub(Z1, add(Z2, Z0)), M2))
    end.
```

```
%%%% divide should not depend on Num1 and Num2,
%thus  Introduce var: Num1 -> Old1, Num2 -> Old2
karatsuba10({Num1, Num2}) ->
    Old1 = Num1,
    S1 = bit_size(Old1),
    Old2 = Num2,
    S2 = bit_size(Old2),
    case is_base(Old1, Old2) of
        true ->
            solve(Old1, Old2, S1, S2);
        false ->
            _ = {Old1, Old2},
            M = max(S1, S2),
            M2 = M - (M div 2),
            <<Low1:M2/bitstring, High1/bitstring>> = Old1,
            <<Low2:M2/bitstring, High2/bitstring>> = Old2,
            SubProblems = [{Low1, Low2}, {add(Low1, High1), add(Low2, High2)},
                {High1, High2}],
            SubSolutions = lists:map(fun karatsuba10/1, SubProblems),
            [Z0, Z1, Z2] = SubSolutions,
            add(add(shift(Z2, M2 * 2), Z0),
                shift(sub(Z1, add(Z2, Z0)), M2))
    end.

%%%%% Extract divide and combine ++ argument reordering if needed
%%%%% Note: extract should ask about the order of the elemnts in
%the returned tuple
karatsuba11({Num1, Num2}) ->
    Old1 = Num1,
    S1 = bit_size(Old1),
    Old2 = Num2,
    S2 = bit_size(Old2),
    case is_base(Old1, Old2) of
        true ->
            solve(Old1, Old2, S1, S2);
        false ->
            {SubProblems, M2} = divide(Old1, Old2, S1, S2),
            SubSolutions = lists:map(fun karatsuba11/1, SubProblems),
            combine(SubSolutions, M2)
    end.

divide(Old1, Old2, S1, S2) ->
    _ = {Old1, Old2},
    M = max(S1, S2),
    M2 = M - (M div 2),
    <<Low1:M2/bitstring, High1/bitstring>> = Old1,
    <<Low2:M2/bitstring, High2/bitstring>> = Old2,
    SubProblems = [{Low1, Low2}, {add(Low1, High1), add(Low2, High2)},
        {High1, High2}],
    {SubProblems, M2}.

combine(SubSolutions, M2) ->
    [Z0, Z1, Z2] = SubSolutions,
    add(add(shift(Z2, M2 * 2), Z0),
        shift(sub(Z1, add(Z2, Z0)), M2)).
```

```
%%%% Generalize function call on is_base -- twice (S1 , S2)
karatsuba12({Num1, Num2}) ->
    Old1 = Num1,
    S1 = bit_size(Old1),
    Old2 = Num2,
    S2 = bit_size(Old2),
    case is_base(Old1, Old2, S1, S2) of
        true ->
            solve(Old1, Old2, S1, S2);
        false ->
            {SubProblems, M2} = divide(Old1, Old2, S1, S2),
            SubSolutions = lists:map(fun karatsuba12/1, SubProblems),
            combine(SubSolutions, M2)
    end.

is_base(Num1, Num2, _, _) ->
    case {Num1, Num2} of
        {<<0:1>>, _} -> true;
        {_, <<0:1>>} -> true;
        {<<1:1>>, _} -> true;
        {_, <<1:1>>} -> true;
        _ -> false
    end.

%%%% Tuple function arguments: is_base, solve, divide
karatsuba13({Num1, Num2}) ->
    Old1 = Num1,
    S1 = bit_size(Old1),
    Old2 = Num2,
    S2 = bit_size(Old2),
    case is_base({Old1, Old2, S1, S2}) of
        true ->
            solve({Old1, Old2, S1, S2});
        false ->
            {SubProblems, M2} = divide({Old1, Old2, S1, S2}),
            SubSolutions = lists:map(fun karatsuba13/1, SubProblems),
            combine(SubSolutions, M2)
    end.

is_base({Num1, Num2, _, _}) ->
    case {Num1, Num2} of
        {<<0:1>>, _} -> true;
        {_, <<0:1>>} -> true;
        {<<1:1>>, _} -> true;
        {_, <<1:1>>} -> true;
        _ -> false
    end.

solve({Num1, Num2, S1, S2}) ->
    case {Num1, Num2} of
        {<<0:1>>, _} -> <<0:S2>>;
        {_, <<0:1>>} -> <<0:S1>>;
        {<<1:1>>, _} -> Num2;
        {_, <<1:1>>} -> Num1
    end.
```

```
divide({Old1, Old2, S1, S2})
 ->
    _ = {Old1, Old2},
    M = max(S1, S2),
    M2 = M - (M div 2),
    <<Low1:M2/bitstring, High1/bitstring>> = Old1,
    <<Low2:M2/bitstring, High2/bitstring>> = Old2,
    SubProblems = [{Low1, Low2}, {add(Low1, High1), add(Low2, High2)},
        {High1, High2}],
    {SubProblems, M2}.

%%%% Introduce Bingings try this one in orden to prove the introducing variable
karatsuba14({Num1, Num2}) ->
    Old1 = Num1,
    S1 = bit_size(Old1),
    Old2 = Num2,
    S2 = bit_size(Old2),
    Bindings = {Old1, Old2, S1, S2},
    case is_base(Bindings) of
        true ->
            solve(Bindings);
        false ->
            {SubProblems, M2} = divide(Bindings),
            SubSolutions = lists:map(fun karatsuba14/1, SubProblems),
            combine(SubSolutions, M2)
    end.

%%%%% Extract function bindings ans tuple arguments
karatsuba15({Num1, Num2}) ->
    Bindings = bindings({Num1, Num2}),
    case is_base(Bindings) of
        true ->
            solve(Bindings);
        false ->
            {SubProblems, M2} = divide(Bindings),
            SubSolutions = lists:map(fun karatsuba15/1, SubProblems),
            combine(SubSolutions, M2)
    end.

%%%% Rename M2 introducing 5 functions

karatsuba16({Num1, Num2}) ->
    Bindings = bindings({Num1, Num2}),
    case is_base(Bindings) of
        true ->
            solve(Bindings);
        false ->
            {SubProblems, Bindings2} = divide(Bindings),
            SubSolutions = lists:map(fun karatsuba16/1, SubProblems),
            combine(SubSolutions, Bindings2)
    end.


%%%% Eliminate variables in bindings
bindings({Num1, Num2}) ->
    {Num1, Num2, bit_size(Num1), bit_size(Num2)}.
```

```
%Parallel version
karatsuba17({Num1, Num2}) ->
    S1 = bit_size( Num1 ),
    S2 = bit_size( Num2 ),
    case is_base(Num1, Num2) of
        true ->
            solve(Num1, Num2, S1, S2);
        false ->
            _ = {Num1, Num2},
            M = max(S1, S2) ,
            M2 = M - (M div 2) ,
            <<Low1:M2/bitstring, High1/bitstring>> = Num1,
            <<Low2:M2/bitstring, High2/bitstring>> = Num2,
            [Z0, Z1, Z2] = pmap(fun karatsubap/1,
                    [{Low1, Low2}, {add(Low1, High1), add(Low2, High2)},
                    {High1, High2}]),
            add(add(shift(Z2, M2 * 2), Z0),
                shift(sub(Z1, add(Z2, Z0)), M2))
    end.

karatsubap({Num1, Num2}) ->
    S1 = bit_size( Num1 ),
    S2 = bit_size( Num2 ),
    case is_base(Num1, Num2) of
        true ->
            solve(Num1, Num2, S1, S2);
        false ->
            _ = {Num1, Num2},
            M = max(S1, S2) ,
            M2 = M - (M div 2) ,
            <<Low1:M2/bitstring, High1/bitstring>> = Num1,
            <<Low2:M2/bitstring, High2/bitstring>> = Num2,
            [Z0, Z1, Z2] = lists:map(fun karatsuba8/1,
                    [{Low1, Low2}, {add(Low1, High1), add(Low2, High2)},
                    {High1, High2}]),
            add(add(shift(Z2, M2 * 2), Z0),
                shift(sub(Z1, add(Z2, Z0)), M2))
    end.

pmap(F, L) ->
    MyPid = self(),
    [spawn(fun() -> MyPid ! F(H)  end) || H <- L],
    [receive
         Res -> Res
     end || _ <- L].
```