Computational Cost Estimation using Volunteer Computing in R

Ricardo Simon Moreira Wagenmaker

Thesis to obtain the Master of Science Degree in Telecommunications and Informatics Engineering

Supervisor: Prof. João Coelho Garcia

Examination Commitee

Chairperson: Prof. Fernando Manuel Bernardo Pereira
Supervisor: Prof. João Coelho Garcia
Member of the Commitee: Prof. José Carlos Alves Pereira Monteiro

October 2017
Acknowledgments

First and foremost I would like to sincerely thank my advisor Professor João Garcia for introducing me to this topic and for his guidance during this whole year, for being always available to help as much as possible, and without whom this work would not have been possible.

I would also like to thank my family who has always inconditionally supported and motivated me throughout this whole process, and to whom I owe my every achievement. Also to my friends, who have always been by my side and kept me going through the hardest times.
Abstract

As organizations realize how data analysis helps them to harness their data and use it to identify new opportunities, the popularity of specialized programming languages like R rises. R is an open source programming language and an environment for statistical computing and graphics, whose increasing notoriety has attracted lots of new users, including everyday users that cannot take the most of the R’s capabilities due to a lack of computing resources. For these reasons, a Volunteer Computing (VC) platform for R software is currently being developed to allow public participants to, voluntarily, share their devices’ idle processing power in exchange for computing credits. These credits can then be used to request for computing power within the platform. In this work we propose a decision system for the mentioned platform that, through estimations, selects the most suitable execution site for a given R script. In order to generate such estimations we follow a history based approach, where we use previous function calls observations to create regression models. The results from this proposed system were validated using the R-Benchmark 25 script, which is globally used in the R community.

Keywords: Volunteer Computing, R programming, Performance Prediction, Computation Offloading, Offloading Decision
Resumo

Com as organizações a aperceberem-se cada vez mais das oportunidades que os dados gerados diariamente fornecem, a popularidade de linguagens de programação especializadas neste âmbito, como o R, tem subido. R é uma linguagem de programação sem proprietário e um ambiente para computação estatística, e cujo aumento de notoriedade tem atraído vários novos utilizadores, o que inclui também aqueles que têm poucos recursos computacionais e por isso não conseguem tirar o máximo proveito das capacidades do R. Por estas razões, está a ser desenvolvida uma plataforma de Computação Voluntária para software em R, que possibilita a qualquer pessoa a oportunidade de, voluntariamente, partilhar os seus recursos computacionais em troca de créditos. Estes créditos podem depois ser usados para pedir recursos computacionais na plataforma. Neste trabalho propõe-se um sistema de decisão para a referida plataforma que, através de estimativas, escolhe qual é o local de execução que providencia mais vantagens. Tais estimativas são obtidas através de um histórico de observações que são usadas para criar modelos de regressão. Os resultados do sistema proposto foram validados usando uma sequência de testes de referência que são usados globalmente na comunidade do R.

**Palavras-chave:** Computação Voluntária, Previsão do Desempenho, *Computation Offloading*, *Offloading Decision*
# Contents

Acknowledgments iii  
Abstract v  
Resumo vii  
List of Figures xi  
List of Tables xiii  
Acronyms xv  

1 Introduction 1  
1.1 Proposed Solution ........................................ 2  

2 Related Work 3  
2.1 Program Performance Analysis .................................. 3  
2.1.1 Timing Analysis ........................................... 4  
2.1.2 Empirical Computational Complexity .......................... 5  
2.1.3 Program Similarity .......................................... 6  
2.1.4 Machine Learning Approaches ................................ 7  
2.1.5 History Based Analysis ...................................... 8  
2.1.6 Discussion .................................................. 9  
2.2 Computation Offloading Decision ................................ 9  
2.2.1 Mobile Computing Context .................................. 10  
2.2.2 Grid Computing Context ..................................... 12  
2.3 Network Bandwidth Estimation ................................ 13  
2.4 R Programming .................................................. 14  
2.4.1 Overview and History of R .................................. 15  
2.4.2 R Interpreter ................................................ 15  
2.4.3 Most Used Packages ........................................ 18
List of Figures

3.1 Formal arguments of function "lm" ................................................................. 21
3.2 Example profile of argument "data" ................................................................. 22
3.3 System architecture .......................................................................................... 24
3.4 Live workflow ................................................................................................. 26
3.5 "qplot" function | table retrieved after full history lookup ............................. 29

4.1 Arguments profile example for function call : lm(weight ~ group, method = "model.frame") 36
4.2 Arguments profile example function call (live prediction): crossprod(Rnorm(2800 * 2800)) 37
4.3 Example code showing independent expressions (left) and it's call tree (right) ........ 37
4.4 Example of a dataset entry's structure .............................................................. 41
4.5 Example of a full history lookup of "qplot" function from package "ggplot2" ........... 43
4.6 Example of a history lookup with argument matching using : qplot(income, education ,
        data=Prestige) ......................................................................................... 43

5.1 Execution estimate vs observed time .............................................................. 48
5.2 Absolute percent error ..................................................................................... 49
5.3 Decision table - predictions ........................................................................... 49
5.4 Decision table - real measurements ............................................................... 50
5.5 Prediction overhead - time ............................................................................ 51
5.6 Prediction overhead - percentage .................................................................. 52
5.7 Execution estimate vs observed time .............................................................. 53
5.8 Absolute percent error ..................................................................................... 53
5.9 Prediction overhead percentage ..................................................................... 54
5.10 Prediction overhead behavior ....................................................................... 56
List of Tables

2.1 Evolution of the most downloaded packages over different periods ................. 19
5.1 External components's information ................................................................ 46
5.2 Results synthesis ......................................................................................... 54
List of Acronyms

VC  Volunteer Computing
WCET  Worst Case Execution Time
ECC  Empirical Computational Complexity
MRE  Mean Relative Error
DTL  Data Temporal Locality
DSL  Data Spatial Locality
ML  Machine Learning
EWMA  Exponentially Weighted Moving Average
RTT  Round-Trip Time
AST  Abstract Syntax Tree
MTBF  Mean Time Between Failures
BW  Bandwidth
SMAPE  Symmetric Mean Absolute Percentage Error
MAD  Mean Absolute Deviation
Chapter 1

Introduction

With the millions of computers of all shapes and sizes that exist all over the world, at any given moment, more than half of the existent computing power is idle. This means that most of the existing processing resources remain unused, when they could be employed, for example, for high performance computing.

This opportunity triggered the interest of researchers, leading to the emergence of Volunteer Computing (VC) as an alternative for computationally intensive projects that do not have the financial power to support the costs. VC is a form of distributed computing which allows public participants to, voluntarily, share their devices’ idle processing power taking advantage of easy-to-use software [1].

Since then, several middleware platforms were designed so that projects could benefit from their capabilities, with BOINC [2] becoming the framework of reference when it comes to building VC projects [3]. These projects are largely situated in the academic research domain, which is the case of SETI@home [4]. Making use of BOINC, this successful project searches massive amounts of radio telescope data for signs of extra-terrestrial intelligence and has gathered over 3 million users.

However, as enticing as this amount of computing power may seem, it cannot be bought, it must be earned. In other words, this means that the projects must have enough public appeal to attract people and get the desired computing power, which does not happen in many other projects. In such projects, although not contributing directly to a greater knowledge, the need for more powerful processing resources is also there.

Along with the increase of the data analysis importance in the business world, as organizations realize how it helps them to harness their data and use it to identify new opportunities, so did the attention given to the R programming language.

R is an open source programming language and an environment for statistical computing and graphics [5], increasingly used by scientists and data miners [6] [7] for developing statistical and data analysis software. An example of a successful project developed with R is Bioconductor [8], which helps scientists to process and compare genetic sequences against databases. The R environment's increasing popularity, together with its open source nature, attracts lots of new users without the financial capacity to support heavy computational power. The problem here is that R based software, like Bioconductor, typically handles large amounts of data while, often resulting in users suffering from lack of computing
resources.

Looking from the regular R user perspective, the resources he has available are often limited to the ones of his computer. Naturally, when running an R script, the intention is for it to take the least time possible to finish, which means using all of his available resources and possibly overloading his local machine with processing. This often turns out to be an issue, as long execution runs may cause users to have to wait for the execution to finish before continuing their work.

In order to fight this shortage of computing power, one could argue that using current VC platforms would aid in solving the problem, but like mentioned earlier, these software development projects do not attract enough volunteer interest.

For these reasons, a VC platform for R software development [9] is currently being developed that allows volunteers to earn credits for sharing their idle computing resources, which can later on be used in the platform when needed. This way people are incentivized to share and become volunteers, since they know they may reap the fruits later.

1.1 Proposed Solution

In such VC platform it is essential to know, a priori, what is the complexity of a certain program. The point is that, if the task is not complex enough, then offloading it would be a waste of time, and being able to predict this makes all the difference. Not only that, but because this platform uses credits as currency, it is necessary to have an estimate of the program’s behavior in order to select the best option out of the available remote volunteers.

The main goal of this dissertation is to present a solution that addresses these requirements, contributing for the development of the mentioned VC platform. The ultimate result is a decision on whether an R program should be offloaded, and if so, select from a pool of remote volunteers which one provides the most benefit. Throughout the document we will refer to this decision as the offloading decision.

Our solution features a combination of execution time estimates with information relative to the remote volunteers, which is obtained through external communication with the VC platform. The decision is carried out by comparing the local execution with the execution on the remote machines, while taking into account the site’s execution time and the estimated credits cost.

Following a history based approach, we use previous function calls observations to create regression models for both their execution times and return object’s memory size. Then, using the available bandwidth and the performance scores correspondent to each remote volunteer available, we determine how much time it takes to execute on those remote machines.
Chapter 2

Related Work

The system that is proposed in this dissertation touches on several domains, which is the reason why this chapter is divided into several sections, providing an overview over some of the major contributions in each area. Each of them has a fundamental role in the final offloading decision.

The decision in our system is going to come up to four major time periods: the time it takes to generate predictions, the program execution time, the time necessary for offloading and the time of getting the results back from the remote site.

The first two periods are approached in Section 2.1, where an insight on some of the techniques used for program performance prediction is provided. Section 2.2 proceeds to point out some of the concerns when performing computation offloading. The offloading time is approached in Section 2.3, which gives an overview over some of the common techniques for measuring available network bandwidth. Finally, Section 2.4 gives a brief explanation of what R programming is, where it stands among other data analysis tools and what are the most used packages.

2.1 Program Performance Analysis

Generally speaking, predicting case scenarios accurately, by getting the best estimates possible, allows to handle them appropriately. This usually provides higher profit and better overall system performance because no resources are wasted.

For this reason the subject of studying the cost of programs and trying to predict the amount of resources needed, although already a well-tried one, is still a popular target for researchers from around the world as they seek out ways to improve scheduling, provisioning and optimization. Example domains include database [10], cluster and cloud [11] and networking [12, 13].

From an R user perspective, the resource that matters the most is time itself. This means that program analysis and the prediction of a program’s execution time assumes a major role in determining if a certain program should be offloaded or not and therefore it will be our focus.

This section presents several approaches that have achieved good results in analyzing the behavior of computer programs, having used one or more of the following techniques: benchmarking, program’s
source code inspection and execution time modeling. This means that these approaches although different in complexity and perspective, have the notion that, in order to obtain a quality prediction, it is necessary to use historical data as a reference and get the most knowledge of the program as possible.

Here we’re looking for techniques that are able to achieve good accuracy while not being too expensive prediction timewise. This means that although accuracy is, of course, one major concern when talking about prediction, in our platform the prediction time (time a technique takes to reach a prediction) is equally important. Otherwise, this could introduce too much overhead in the system and cancel all the benefits that offloading the computation would bring.

2.1.1 Timing Analysis

Timing analysis refers to the process of discovering execution-time bounds or estimates [14]. To avoid having to deal with the halting problem [15], this technique is usually done in the context of real-time systems, which use a restricted form of programming and guarantee the termination of its programs. This type of programs have hard timing constraints and their runtime may never or almost never exceed the upper bounds previously determined, depending on whether they belong to a hard real-time system [16] or not. It is imperative that the defined constraints are respected as these systems are designed while taking into account these values.

The goal of this process is to come up with a value that represents the longest execution time possible of a certain program, which is called the worst-case execution time (WCET). This way it is possible to show whether a certain program respects the desired time constraints or not.

There are two major approaches for performing timing analysis:

– **Static.** This technique obtains the WCET by analyzing, from the task’s code, the number of possible paths that the task to be executed may take and combines this knowledge with a model of the hardware architecture.

– **Measurement-based.** This approach executes the task for some arbitrary input, on hardware or on a simulator. They obtain an estimate of the WCET, which comes from measuring the task’s end-to-end execution time for a subset of the possible execution paths. In this case the WCET estimate takes the time value of the longest execution observed.

There are several timing-analysis tools which follow the static approach [14], but this approach suffers from a major disadvantage: it is very difficult to accurately model both the hardware architecture and the software environment [17]. More so given that the complexity of the hardware features keeps increasing. In addition to these technical disadvantages, which every approach has, these problems are emphasized when trying to use a tool with a different processor, since adapting a static analysis tool to new hardware is extremely expensive. There have been efforts to address these problems, like PROARTIS [18], but the added trouble to ensure safety and get real, safe bounds instead of estimates doesn’t make sense in the scenario of this project, since it does not have hard real-time characteristics. Using a measurement-based approach is effectively more adequate for this project’s context. Here the values of the boundaries
would represent decision reference levels, which don’t need to be absolutely observed. For example, if the estimated WCET is inferior to a predefined offloading threshold, then it is very likely that the computation should be done locally.

Despite the fact that timing analysis gives valuable information about the program’s behavior when executed, it cannot be used by itself to decide whether it should be offloaded to the volunteer computing platform or not. This is so because the knowledge taken from performing this analysis is only about the worst case scenario and the decision process requires a more detailed analysis centered on the critical scenario, where the decision transits from A to B.

2.1.2 Empirical Computational Complexity

Another way of increasing the knowledge about a certain program is by modeling its asymptotic behavior using information provided from execution traces. The asymptotic behavior of a program refers to the growth of the program’s execution time as the size of the input increases and is typically analyzed when there is the need to estimate how complex a program will be on large inputs.

Trendprof [19] does exactly that, and proposes the notion of empirical computational complexity (ECC) for describing the scalability of a program. The Trend Profiler (trend-prof) is a tool that is able to construct ECC models, which have the capacity to provide information with a great level of detail and small granularity about the behavior of a program. This not only gives a great perspective about what is going on within the execution, but also brings to the table different prediction possibilities.

The tool constructs these models by observing multiple runs of the program, measuring how many times each basic block in a program is executed and inserting this information in a linear \((y = a + bx)\) and power law \((y = ax^b)\) models. These models reveal performance trends which can be used to extrapolate and predict how many times each basic block in a program is executed, depending on a user-specified input size.

They were able to get impressive results with some programs, getting less than 5% of mean relative error (MRE) when fitting the observations to the models, but with other programs the MRE was as high as 140%. However, whenever there is a trend in the observations, the tool is able to detect it and in such cases the maximum MRE recorded was of 30%. Besides that, they concluded that programs mostly scale as a power law.

These results give us confidence that this tool can be used to detect trends in the performance scalability of a program, although not being completely reliable if one pretends to get a specific performance value for a chosen input size.

Although here, the measure of performance is not time, but basic block counts, if the approach is extended to associate a time value to each basic block, then it is possible to reach a prediction of a certain program’s execution time.
2.1.3 Program Similarity

Measuring program performance from a benchmark suite is not, by all means, a new subject in computer architecture research [20]. A more recent approach though, is to reduce the benchmark suite used so that has only the benchmarks programs most similar to the program being measured [21]. By introducing this new component of measuring the similarity between the new program and the benchmark programs, it is possible to quickly predict performance without running long simulations.

This similarity measurement is done using microarchitecture independent metrics, from which a subset was used. These can be classified into five categories:

- **Instruction mix.** Measures the relative frequency of various operations performed by a program.

- **Branch behavior.** Knowing that a branch is an instruction that can lead a computer to start executing a different instruction sequence, a backward branch is one that has a target program counter lower than its own. The backward branches are more likely to be taken than their forward kind.
  
  - **Branch Direction:** Percentage of forward branches out of the total branch instructions.
  
  - **Fraction of taken branches:** Ratio of taken branches to the total number of branch instructions.
  
  - **Fraction of forward-taken branches:** Ratio of forward branches taken to the total number of forward branches possible.

- **Inherent instruction level parallelism.**
  
  - **Basic Block Size:** Average number of instructions between two consecutive branches.
  
  - **Register Dependency Distance:** Total number of instructions between a write and read of a register instance.

- **Data locality.**
  
  - **Data Temporal Locality (DTL):** Average number of memory accesses between two consecutive accesses to the same address, per address in the program. The evaluation is performed with four cache block sizes (16, 64, 256 and 4096 bytes).
  
  - **Data Spatial Locality (DSL):** Ratio between the DTL metrics for cache block sizes of 64, 256 and 4096 and the DTL for cache size 16.

- **Instruction locality.**
  
  - **Instruction Temporal Locality (ITL):** Average number of memory accesses between two consecutive accesses to the same static instruction, for every instruction in the program that is executed at least twice. The evaluation is performed with four cache block sizes (16, 64, 256 and 4096 bytes).
– **Instruction Spatial Locality (STL):** Ratio between the ITL metrics for cache block sizes of 64, 256 and 4096 and the ITL for cache size 16.

The use of these metrics allows a comparison between programs free from features specific of particular microarchitectural components, leading to reliable results without having to consider the conditions on which the benchmarks were generated.

The two methods proposed in [21] for this approach are the following:

1. Assign weights to each of the benchmark programs based on their similarity to the new program and then use the weighted mean to predict the performance.

2. Cluster similar benchmarks and predict performance using the representative program from the cluster which contains the new program.

The first method, while using the weighted harmonic mean, was able to achieve a worst case prediction error of 11.89%. The second method achieved an average prediction error of 20.29%. Both errors are relative to the prediction of the speedup of a new program, using the SPECint2000 [22] benchmark programs.

### 2.1.4 Machine Learning Approaches

In order to accurately predict program execution time on a new input, large amounts of data must be taken into consideration. It is a long time desire of computing researchers to perform such prediction, but nowadays it is indeed possible to do just that. By using machine learning techniques to model the algorithm’s runtime as a function of specific features correlated with the program’s execution time, it is possible to achieve great results in the performance prediction domain. Machine learning (ML) is a very vast field of study which includes several methods of data analysis that automate analytical model building. Using algorithms that continuously learn from data, it is possible to have computers effectively learning hidden patterns and making predictions based on that data. This has become the go to approach when talking about analysis of big amounts of data, as it continues to show how valuable it can be.

There are several works that use machine learning to predict performance that have obtained good results [23], but one framework in particular was able to achieve impressive results in their prediction - Mantis [24, 25], which is why this section focuses on it.

Mantis is a framework for predicting program performance on given inputs. It does so on an automatic manner, while being both accurate and efficient, by combining techniques from program analysis and machine learning. This framework is particularly interesting for our platform, because unlike the approaches presented so far, it aims for an algorithm that doesn’t introduce too much predicting overhead. It divides its procedure in two stages: online and offline. The online stage can be easily compared to the analysis of a program and its input at execution time, leading to the decision about whether the said program should be offloaded or not.
In order to make an efficient online stage, the offline one has to be carefully designed, which in this framework consists of four components:

- **Feature Instrumentor.** Analyses the program’s code, adds instrumentation code that makes possible the collection of program features relevant to performance. These features must capture the program performance’s behavior while being accurate, easy to compute and introduce low overhead. The following features were used: *Loop Counts, Branch Counts, Method Invocation Counts* and *Variable Values*.

- **Profiler.** Runs the instrumented program with sample input data and sends tuples of (execution time, feature values) to the model generator.

- **Performance Model Generator.** Relies on machine learning techniques, more specifically, on sparse nonlinear regression. Although efficient algorithms have been developed for simultaneous feature selection and model fitting, like LASSO and FoBa, the authors of Mantis used a solution developed by them that outperforms both of the mentioned algorithms [26]. Using the SPORE-FoBa algorithm [26], this component of the offline process filters out many features that are not useful to the execution time analysis and captures the nonlinear combination of the relevant features.

- **Predictor Code Generator.** Using *program slicing* [27, 28], it computes, for each given feature, the set of all the statements in the program that have a chance of affecting the feature’s value. Each slice is an executable program, which, when executed with a sample input, provides both the evaluation cost and the value of the respective feature.

This design allows Mantis to achieve less than 5% of prediction error in most cases, and very notably, a prediction time with a maximum of 1.3% of the original execution time.

Although recent frameworks using machine learning techniques to predict performance have revealed good results, this modeling approach may reveal itself to be too expensive in the context of our project, considering that developers, while creating a program, keep changing and testing its code iteratively until the final product is obtained. So if one would create a prediction model each time the code is modified, the efficiency obtained would not be optimal.

### 2.1.5 History Based Analysis

Besides the already mentioned approaches to performance prediction, there are simpler ones that are purely based on previously recorded observations [29]. Surprisingly, although simple and in some cases even naive, these forecasting methods sometimes reveal themselves to actually be the best option to model the events.

Some of the most used are :

- **Average.** The next forecast value is set to be equal to the mean of the observations. There are different types of averages and, although the most common is the arithmetic mean, there are other solutions that are less naive. Examples of that are the weighted mean, where the observations
can be given weights according to the context they are in, or even the moving average, where only the last $x$ observations are used for the calculation.

- **Naïve.** All forecasts are simply set to be the value of the last observation.

- **Drift method.** A variation of the Naïve method, where the forecasts increase or decrease over time depending on the context the users apply it on. The drift, is set to be the average of the changes seen in the historical data.

Projects that pretend to reduce the complexity of their system sometimes utilize these or similar simple predicting techniques, accepting and dealing with the error they may introduce in the calculations. One such project is Phone2Cloud [30], which is a computation offloading-based system for energy saving on smartphones in the context of mobile cloud computing. Phone2Cloud, in order to predict the execution time of an application running on the smartphone, executes the steps shown below:

1. Gets the input size of the application and gets the predicted average CPU workload from the resource monitor.
2. Uses the input size and the average CPU workload to search a log for the two nearest points in Euclidean distance [31].
3. Returns the mean of the two points’ execution times as the predicted value.

Here, the log is the execution history log in the fixed form (Application, Input size, Average CPU workload, Execution time) and, since the log entries of a specific application are seen as points in a three-dimensional space (Input Size, Average CPU Workload and Execution Time), the distance between points can easily be computed.

### 2.1.6 Discussion

One of the major concerns referred in the beginning of Section 2.1 is the prediction overhead each technique may introduce in the system, as too much of it cancels all the benefits that offloading the computation would bring. Although of noted importance, it was not given emphasis when describing each technique because the prediction overhead each of them generate is not mentioned in their respective literature.

Actually, the only work that had this concern was Mantis, described in Section 2.1.4. This is probably due to the fact that most of the presented techniques focus their efforts on achieving the best prediction possible in an offline mode and are not concerned with their integration in a system with real time interaction.

### 2.2 Computation Offloading Decision

Computation offloading is the execution of parts or of a whole program code on the cloud or on specific surrogates [32]. The execution may be resource intensive (e.g: battery, memory, processing power) or
computation intensive (processing power), being that the latter is employed when one wants to specify that the resource required is processing power.

This concept resulted from the increased usage of resource constrained devices, providing a solution to the limitations these devices present. Being processing power, battery power or storage capacity aware, it increases the usability of such devices, like laptops or mobile phones, regardless of their physical limits. For this reason, offloading processing to more powerful computers has been a hot topic among computing researchers. Although this resource saving technique is usually associated with mobile computing, it can also be used in schedulers of computational grids [33], helping to improve overall performance.

One question that is inherently related to the aforementioned computation offloading, is when to perform it. One can easily understand that such a decision should only be taken if the additional performance achieved when offloading is not overshadowed by the cost it takes to move the input and the program's code to the remote location and then retrieve the results.

When comparing the two contexts mentioned earlier in which this technique may be used, their focus has one fundamental difference: battery saving. Whereas in a mobile context, one of the main goals is reducing energy consumption and extending the battery duration [30,34], in grid computing, the goal is solely improving performance, i.e., reducing the time required for execution. At the end of the day, the final call will always depend on what is the focus of each project, and what factor they value more.

One factor that affects every offloading decision, is the available network bandwidth. Design choices may lead some systems to not consider it and assume bandwidth to be constant, like is the case of static partitioning approaches, but most of the times that's not the case. In this type of approach the code that may be offloaded is decided before executing the program itself, which means that the partitioning is done at the time of development.

As much as one would like to take "available bandwidth" out of the equation, in real systems it is variable and has a lot of influence in the performance and satisfaction of users. Considering this, we now focus on two frameworks that take bandwidth into account in their offloading decisions, showing the impact it has in both the mobile computing context and the grid computing one.

### 2.2.1 Mobile Computing Context

The paper [34] describes an offloading system that focuses on improving the static partition approach by considering bandwidth change in the decision making. Since we are speaking of a mobile computing context, "local" will mean that the execution is performed on the mobile device. Their algorithm has four major components:

- **Construction of Weighted Object Relation Graph.** Constructs a relation graph by combining static analysis and dynamic profiling [35]. The graph has class level granularity, which means that every graph node represents a class and has associated a value of 1 or 0, if the node can be offloaded or not, respectively.

- **Time and Energy Weighted Optimization Model.** Models both time (T) and energy (E) required
for a certain partition solution \( S \) and a given bandwidth \( B \). These models are both optimized, which means that every node that can be executed remotely is assumed to be executed there.

\[
T(S, B) = T_{\text{local}} + T_{\text{remote}} + T_{\text{communication}} \tag{2.1}
\]

\[
E(S, B) = E_{\text{local}} + E_{\text{communication}} \tag{2.2}
\]

Equation 2.1 is relative to time calculation, and it gives an optimized execution time estimate. The result comes from the sum of the duration times of both local \( T_{\text{local}} \) and remote \( T_{\text{remote}} \) execution, plus the time of communication between local and remote nodes \( T_{\text{communication}} \).

Equation 2.2 is relative to energy calculation, and gives an optimized energy consumption estimate. The final value comes from the sum of energy used duration local \( E_{\text{local}} \) execution and the energy necessary for the communication interfaces \( E_{\text{communication}} \).

\[
W(S, B) = \left( \frac{T(S, B)}{T_{\text{local}}} \right) \times w_t + \left( \frac{E(S, B)}{E_{\text{local}}} \right) \times w_e \tag{2.3}
\]

Next the cost obtained with Equations 2.1 and 2.2 are combined with Equation 2.3. Here both costs are weighted, giving an importance value to saving time and saving energy. In this offloading system this weight assignment is done by the users, and as long as they respect the condition \( w_t + w_e = 1 \), they are given the choice of what weight values should be given to saving time \( w_t \) and saving energy \( w_e \). This way the users can express their preferences and indicate what should be the focus of the system while deciding if it is worth it to offload the computation.

- **Threshold Calculation.** The cost threshold is equal to the cost obtained if nothing is offloaded, while using the time and energy weight model explained above.

- **Partition Decision.** Divides the graph into subgraphs and using the cost threshold defined earlier, takes decisions for each of the subgraphs. This means that, starting from the node farthest from root node, defines which of the subgraphs can be offloaded so that the partition remains efficient, i.e. so that the total cost remains below the threshold.

Using this algorithm they are always able to find a solution that is efficient, optimizing time and energy when compared to normal application execution. Moreover, it is observed that execution time and energy consumed are inversely proportional to available bandwidth, proving that it must be taken into account when performing quality offloading decisions.

In this system the focus was in modeling the offloading cost without disregarding the available bandwidth, and so, how the values of both time and energy estimates are obtained is not explained. Also, the measuring of the available bandwidth is not described, but it is expected that the bandwidth estimation uses a similar technique to those referred in Section 2.3.
2.2.2 Grid Computing Context

The paper [33] presents a framework for making computation offloading decisions in a computational grid. This system approaches the problem in a statistical fashion, trying to predict when an offloaded computation outperforms the local one.

In computational grids, schedulers must be capable of managing the available resources in the best way possible so that performance is optimized. In this context, energy saving is not a priority and so it is not even considered in the offloading decision, which is now purely based on reducing the time it takes to execute a certain task. Besides that, there is no partitioning of the task, so the scheduler must take into account that if a task is to be executed remotely then its whole context must be moved.

Considering the context, they observed that the offloading decision comes down to three predictions:

1. Time required to execute the computation on the local machine ($C_l$).
2. Time required to execute the computation remotely, once all the necessary data is successfully received at the remote site ($C_d$).
3. Time required to move the task’s code and its input data to the remote site and gather the results back at local site after the execution.

In this work, the focus of the authors was in getting the best estimate possible for the expected offloading performance and consequently make better offloading decisions. Their intent was to reach such a goal by maximizing the accuracy with which the network bandwidth predictions were performed, and so the first two predictions were assumed to be known by the scheduler.

In order to do so, they start by creating a cost model of the task execution. If the speed-up of $C_d$ relative to $C_l$ and the network bandwidth are given by $\alpha$ and $\beta$ respectively, and the amount of data to be moved is given by $s$, then the expected cost of remote execution $C_r$ can be computed as:

$$C_r(\beta) = C_l \alpha + \frac{s}{\beta} \tag{2.4}$$

And since computation offloading is only beneficial if $C_l > C_r(\beta)$, it is possible to determine the minimum bandwidth necessary for the offloading to go forward, defined as critical network bandwidth:

$$\beta > \frac{s}{C_l - C_l \alpha} \tag{2.5}$$

Using this model and a Bayes decision strategy that incorporates Change-point detection as a method to predict the state of the network bandwidth during offloading, this framework was able to perform accurate offloading decisions and reduce the amount of overhead resultant from the times the scheduler makes the wrong offloading decision.

The Bayesian decision strategy introduces the notion of the risk of fully believing in the value the network predictor comes up with. This approach allows the computation of the expected risk.
2.3 Network Bandwidth Estimation

As mentioned earlier in section 2.2, network bandwidth estimation assumes a major role in the offloading decision, being most of the times the deciding factor. Producing quality estimates is challenging as, besides the fluctuating nature of the network bandwidth, the occurrence of observation noise becomes a lot of times a disrupting factor. This observation noise is referred as the measurement error introduced by events that are not accounted for in the measurement, like routing changes or network congestion. Since the available bandwidth is such an important information to have in computing systems, several works were done in an attempt to reach the best estimate possible.

Most of the existing systems to predict network performance can be divided in two techniques:

- **Active.** Injects traffic in the network solely for measuring purposes.

- **Passive.** Uses traffic already present in the network for observing its behavior.

There are several approaches that use the active probing technique proposed by Keshav in [36], like Bolot [37], for example, which uses pairs of UDP packets to determine the network state. Or Pathchar [38], which uses ICMP packets with varying TTL fields, but these tend to suffer from heavy bandwidth consumption, which in cases where the network bandwidth is low is not an option.

More lightweight systems have been developed using passive measurements. Such are the cases of the round-trip time estimator in TCP [39] and of the Odyssey platform [40]. These two approaches use an *exponentially-weighted moving average* (EWMA) to filter the noisy observations and get a network bandwidth estimate value.

EWMA is a filter that, given a new observation, produces a new estimate as a linear combination of the last computed estimate plus the new observation. In traditional implementations of this filter a fixed weight is given to both, and depending on what has more weight, different design goals may be achieved:

- **Stability.** The system has a better resistance to noisy observations

- **Agility.** The system is capable of more quickly detecting bandwidth variations.

In TCP’s RTT estimator, old estimates are given more weight in an attempt to get better stability and not let the estimate of the available bandwidth be easily influenced by performance variations in the network. On the other hand, Odyssey’s estimator gives more weight to the new observations in order to quickly track changes and allow applications to make better decisions.

Although these systems work well in their context, neither property is advantageous all the times. In the paper [41] was proposed a filter, *Flip-flop*, that can be stable or agile, trading stability for agility, depending on what are the conditions at the time, which means that it adapts to the circumstances.

This work [41] starts by passively probing the network (without adding measurement overhead), and measuring round-trip time (RTT) in order to represent the behavior of packets along the end-to-end path between two machines. Every time an observation is performed, the current bandwidth is inferred and then inserted in the EWMA filter (Equation 2.6) to reduce the possible noise it may have.
\[ E_t = \alpha E_{t-1} + (1 - \alpha)O_t \] (2.6)

In Equation 2.6, \( E_t \) represents the new estimate, \( E_{t-1} \) is the last estimate computed and \( O_t \) is the current observation. The gain, which is represented by \( \alpha \), determines the filter’s reactivity. So, the higher the gain, the slower it reacts to network performance changes, making the system stable.

The adaptability of the Flip-Flop filter comes from having two of these EWMA filters, one stable (gain is 0.9) and one agile (gain is 0.1), and having a controller selecting between the two. The principle is to select the agile filter whenever it is possible, but select the stable one if unusually noisy observations come up. This means that the stable filter is selected every time the previously defined control limits (typically three times the sample standard deviation) are exceeded.

In another work, the forecasting component of the Network Weather Service, NWS [42], introduced a different perspective on the subject. Here, instead of relying on a single model, this forecasting system simultaneously exercises a set of models to generate predictions for every measurement value observed in a history window, being that every prediction can only be based on measurements that come before it. After that, each forecast and its respective measured value are compared and each model is assigned an overall accuracy score and the one with the highest provides the chosen bandwidth prediction.

Other simpler prediction methods, like using the last observation as a forecast for the next measurement or performing a simple mean of the observations and using the average value as a estimate, are commonly used to compute a history based prediction. Although seemingly a little naive, with the right decision strategy, these actually perform pretty well when used in the offloading decision process [33].

2.4 R Programming

R is an open source programming language and an environment for statistical computing and graphics [5], being increasingly used by scientists and data miners [6] [7] for developing statistical and data analysis software. It has established itself as the most-used data mining tool and the number of analysts that are using it as a primary tool is growing every year [43].

This is supported by the number of successful projects developed under the R environment like is Bioconductor [8], which helps scientists to process and compare genetic sequences against databases, or Rmetrics [44], which is an open source solution for financial market analysis and valuation of financial instruments, or even Spatial Data Science [45], which develops software for geospatial analysis and geo-visualization under R.

In this section we start by giving an overview of the design and history of R, and follow it by focusing on the R’s language definition and how the R interpreter translates a given R script into executable instructions. Afterwards, we also present which were the most downloaded packages from the official repositories.
2.4.1 Overview and History of R

R is a different implementation of the S language [46], which was designed in the 1980s and has been in widespread use in the statistical community since. Although there are important differences, much of the code written for S runs unaltered under R. It is a GNU [47] project, which means that it is free software, i.e., users have the freedom to run, copy, distribute, study, change and improve the software.

As noted by the language's original creator in [48], the basic idea of the S language, and later the R language, was that people would enter the language in an interactive environment. Here they could use the language and the environment, without requiring previous programming knowledge or having to know very detailed aspects of the language. They could use the environment to look at data, and do basic analysis. And then as their needs became clearer and their sophistication increased, they could be able to seamlessly get into programming, where the language and system aspect would become more important.

The R system is divided into two conceptual parts, having its functionality divided into modular packages.

1. The "base" R system from the official repository CRAN [49]
2. Every other package

The base R can be further divided into the base package, and the ones that provide extra functionality. The core of the R system, as programming language, consists in a package named "base". It is required in order to run R and contains the most fundamental functions, including essential aspects as language interpretation and some basic functionality. Then there are others packages which provide the remainder of R's base functionality: utils, stats, datasets, graphics, grDevices, grid, methods, tools, parallel, compiler, splines, tcltk and stats4.

Finally, there are other packages that are not included in the basic installation of R. The R functionality can be easily extended, either by packages that users create themselves to suit their needs, or by adding packages from online repositories, where users share their contributions in a worldwide community. The biggest repository is CRAN, which is the official repository and currently has over 11000 available packages [50]. Out of those, there are a few that are recommended by the R core team, being updated on CRAN upon each new release of R. These are: boot, class, cluster, codetools, foreign, kernsmooth, lattice, mass, matrix, mgcv, nlme, nnet, rpart, spatial, survival [51].

2.4.2 R Interpreter

When a user types a command at the prompt (or when an expression is read from a file) the first thing that happens is that the command is transformed by the parser into an internal representation. The evaluator executes parsed R expressions and returns the value of the expression. All expressions have a value. This is the core of the language [52].

There are three types of objects that constitute the R language: calls, expressions, and names. Since R has objects of type "expression", we make the distinction between expression and statement
terms, using the latter for syntactically correct expressions.

A statement, such as \( x <- 1:10 \) or \( \text{mean}(y) \), is a syntactically correct collection of tokens, which are the elementary building blocks of a programming language (constants, identifiers, reserved words, separators, operator symbols, grouping and indexing symbols). Expression objects are special language objects, which contain parsed but unevaluated R statements. The main difference is that an expression object can contain several such expressions.

Every elementary R expression is represented internally in function call form, as a list with the first element containing the function name and the remainder containing the arguments, which may in turn be further R expressions, e.g. the assignment \( x <- 1 \) is encoded as "\( \sim \)(x, 1). Note that this is valid not only for expressions, but for all R syntax elements.

An R program consists of a sequence of R expressions. An expression can be a simple expression consisting of only a constant or an identifier, or it can be a compound expression constructed from other parts (which may themselves be expressions). There are six possible syntactical constructs an expression may take:

- **Function calls.**
  
  A function call takes the form of a function reference followed by a comma-separated list of arguments within a set of parentheses: \( \text{function reference}( \text{arg1}, \text{arg2}, ..., \text{argn}) \).
  
  The function reference can be either an identifier (the name of the function), a text string or an expression (which should evaluate to a function object).
  
  Each argument can be tagged (tag=expr), or just be a simple expression. It can also be empty or it can be a special token. A tag can be an identifier or a text string.
  
  Example: \( g(5, 1+1, \text{tag} = \text{value}) \) is encoded as \( g(5, 1+1,\text{tag} = \text{value}, ) \)

- **Infix and prefix operators.**
  
  Except for the syntax, there is no difference between applying an operator and calling a function.
  
  Example: \( x + y \) is encoded as \( + (x, y) \).
  
  Notice that since \( + \) is a non-standard function name, it needs to be quoted.

- **Index constructions.**
  
  R has three indexing constructs, two of which are syntactically similar although with somewhat different semantics:

  - \( \text{object} \{ \text{arg1}, ..., \text{argn} \} \)
  
  - \( \text{object} [\{ \text{arg1}, ... , \text{argn} \}] \)
  
  - \( \text{object} \$ \text{tag} \)

  Whereas the last item requires the "object" to be a subsetting object, in the first two constructs, the "object" can formally be any valid expression, as long as it denotes or evaluates to a subsetting object. The arguments generally evaluate to numerical or character indices.
Internally, these index constructs are stored as function calls with function name ‘[’, ‘[[’ and ‘$’, respectively.

– **Compound expressions.**

A compound expression is a sequence of expressions, separated either by semicolons or by newlines.

Example: \{expr1; expr2; expr3; expr4\} is encoded as ‘[’ (expr1, expr2, expr3, expr4)

– **Flow control elements.**

R contains the following control structures: if, is..else, while, repeat and for.

Their special syntactic constructs are encoded the following way:

– if (cond) expr -> "if"(cond, expr)
– if (cond) expr1 else expr2 -> "if"(cond, expr1, expr2)
– while (cond) expr -> "while"(cond, expr)
– repeat expr -> "repeat"(expr)
– for (var in list) -> "for"(var, list, expr)

– **Function definitions.**

Note that as mentioned earlier, every construct has the same internal representation, and therefore accessed the same way. The first element, or in other words, the "head" of the list, works as a function name and indicates how the interpreter should act next. The remainder of the elements, or the "tail" of the list, are used as arguments for the next evaluating step.

**Functions**

In R, functions are objects and can be manipulated in much the same way as any other object. Every object can be characterized in one following three function types:

– **Closure.** A regular function
– **Special.** An internal function that does not evaluate its arguments
– **Builtin.** An internal function that evaluates its arguments

Function closures have three basic components: a formal argument list, a body and an environment.

The formal argument list is a comma-separated list of arguments. An argument can be a symbol, or a ‘symbol = default’ construct, or the special argument ‘...’. The second form of argument is used to specify a default value for an argument, which is used if the function is called without any value specified for that argument. The ‘...’ argument is special and can contain any number of arguments. It is generally used if the number of arguments is unknown or in cases where the arguments will be passed on to another function.
The body is a parsed R statement. It is usually a collection of statements in braces but it can be a single statement, a symbol or even a constant. A function’s environment is the environment that was active at the time that the function was created.

The first thing that occurs in a function evaluation is the matching of formal to the actual or supplied arguments. Both Special and Builtin functions typically ignore tags and do positional matching (explained below), with some exceptions.

Closure functions go through a three step process:

1. **Exact matching on tags.** For each named supplied argument, the list of formal arguments is searched for an item whose name matches exactly. It is an error to have the same formal argument match several actuals or vice versa.

2. **Partial matching on tags.** Each remaining named supplied argument is compared to the remaining formal arguments using partial matching. If the name of the supplied argument matches exactly with the first part of a formal argument then the two arguments are considered to be matched. It is an error to have multiple partial matches.

   Notice that if \( f \leftarrow \text{function}(\text{fumble}, \text{fooey}) \ f\text{body} \), then \( f(f = 1, fo = 2) \) is illegal, even though the 2nd actual argument only matches \( \text{fooey} \). \( f(f = 1, \text{fooey} = 2) \) is legal though since the second argument matches exactly and is removed from consideration for partial matching. If the formal arguments contain ‘…’ then partial matching is only applied to arguments that precede it.

3. **Positional matching.** Any unmatched formal arguments are bound to unnamed supplied arguments, in order. If there is a ‘…’ argument, it will take up the remaining arguments, tagged or not.

   If any arguments remain unmatched an error is declared.

**2.4.3 Typical Use of R**

When performing data science, which is often done under the R environment, there is the regular need to repeat variations of large analysis to learn, infer parameters and come up with an estimate of the model stability. This prompts analysts to seek a more efficient computing model and use packages like **parallel** and **foreach** to take advantage of parallel computing in R, in which many calculations are carried out simultaneously.

One characteristic of these computations is that they are a lot of times perfectly parallel, i.e. they are independent and do not need to communicate in any way. This represents an opportunity for the platform proposed in this work and to consider offloading different chunks of computation to different volunteers. Of course it would be necessary to estimate the complexity of each individual computation, which would create additional estimation overhead, but it should be considered as it may greatly improve the global performance.
Table 2.1: Evolution of the most downloaded packages over different periods

<table>
<thead>
<tr>
<th>Rank</th>
<th>Package 1</th>
<th>Package 2</th>
<th>Package 3</th>
<th>Package 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>plyr</td>
<td>Rccp</td>
<td>ggplot2</td>
<td>scales</td>
</tr>
<tr>
<td>2</td>
<td>digest</td>
<td>ggplot2</td>
<td>Rccp</td>
<td>scales</td>
</tr>
<tr>
<td>3</td>
<td>ggplot2</td>
<td>stringr</td>
<td>colorspace</td>
<td>tibble</td>
</tr>
<tr>
<td>4</td>
<td>stringr</td>
<td>colorspace</td>
<td>digest</td>
<td>stringr</td>
</tr>
<tr>
<td>5</td>
<td>RColorBrewer</td>
<td>digest</td>
<td>reshape2</td>
<td>colorspace</td>
</tr>
<tr>
<td>6</td>
<td>reshape2</td>
<td>reshape2</td>
<td>RColorBrewer</td>
<td>stringr</td>
</tr>
<tr>
<td>7</td>
<td>proto</td>
<td>manipulate</td>
<td>mime</td>
<td>digest</td>
</tr>
<tr>
<td>8</td>
<td>scales</td>
<td>mime</td>
<td>digest</td>
<td></td>
</tr>
</tbody>
</table>

2.4.4 Most Used Packages

One of the major characteristics of R is the possibility it provides to developers to easily extend its functionalities by creating additional packages for specific researches or with global functionalities that may be applied to several projects. This leads to several scientists and analysts using packages already developed by other people that had the same problem or concerns.

In order to compare the most popular packages, we show in Table 2.1 the top 10 of the most downloaded packages in the months of January until May in 2013 [53], in April of 2015 [54] and November of 2016. The data from the first two periods was taken from the CRAN [49] repository alone, but the last one was taken from Rdocumentation.org [29] which aggregates R documentation and download information from the most popular repositories like CRAN, BioConductor and GitHub.

While analyzing Table 2.1, it is possible to see that through the years there has been little change in the most used packages, if the number of downloads is employed as usage metric. The one eye-catching change is the sudden appearance of package Rccp on top of the list. This matches the aforementioned tendency of using R as a primary tool for data analysis, as analysts want to write high performance code in R and use the Rccp package to obtain a seamless integration of R and C++.

Another change, yet more subtle, is the rise to the top of package ggplot2, which offers an easy, consistent interface for producing plots and charts in R. This development was expected, since visualizing the results is the best way data analysts have to take conclusions.

Although the tendency to use R in producing high performance tools may provoke the use of more complex packages, it is possible to see that most of the most used packages in the beginning of 2013 are still used in November of 2016. This means that simple packages like stringr (which provides simpler and more consistent functions for string processing than those present in R), are still important in the regular code development under the R environment.

This analysis of the most used packages may help in predicting the complexity associated with the execution of an R program as it allows to introduce additional information in the process about packages that are probably going to be used. If, for example, one package is labeled as complex and is present in a program, when the offloading decision is performed, the offload decider will have an additional hint.
Chapter 3

Predictor System

In this chapter we describe a system capable of deciding whether an R program should be offloaded, and if so, select from a pool of remote volunteers which one provides the most benefit. This chapter starts with an overview of the system’s design, followed by the architecture of the solution, in which we take a closer look at each of the architecture’s components, their workflow and the techniques they employ.

This work aims to provide a major addition to the volunteer computing project in which it belongs to and develop a prototype of a system capable of performing the offloading decision.

This can be accomplished by following a history based approach, using previous observations of function calls to reach an estimation value. The decision is carried out by combining the information resulting from the algorithm mentioned above, and both the available bandwidth and the performance score correspondent to each remote machine available. In order for the system to provide accurate results, it needs to have at its disposal an adequate dataset of observations.

Our Predictor System comes to help resolving a few fundamental needs of an R user:

1. **A user intends not to overload its own computer with processing.** So that she may keep using it while the processing is done, the user wants to have the program execute elsewhere, giving emphasis to saving local RAM and CPU.

2. **A user wants to get faster results out of the program it is running.** Here the emphasis is on obtaining extra processing power, so that the program can be computed at a higher rate than the one she currently has.

3. **A user wants to extend their RAM.** Here the user wishes to add extra RAM so that the program may load bigger datasets into memory.

Naturally, these items are not exclusive, i.e, it is possible that a combination of the three happen. In this work though, we do not address the third item, instead the system loads the datasets into the environment just like R currently does.
3.1 Approach Overview

The core of the prediction system presented in this work is estimating the execution time of a certain expression in R. When talking about how the system is able to succeed, this is the real challenge.

As seen in the previous chapter (Section 2.4.2), every operation in the R programming language is analysed as a function call. Moreover, not only do almost all of these functions belong to a package in CRAN, but the R interpreter accesses them all similarly, i.e., it doesn’t matter if they are from the base packages or belong to an external package, they are all handled the same way.

Given this, our approach to generate an execution time prediction consists in using previously made profiles of both the closure and builtin functions (Section 2.4.2) to generate a final estimate value. A function operates in its own environment, where most of the times the only variables defined at the start are the arguments it received upon being called. This means that the behavior of the function directly depends on what arguments it receives, and whatever that behavior might be, it varies depending on the various arguments’ value, type, size, class, to name a few features. But it is important to note that not all of the arguments have the same influence. By looking at a function call we can usually see two cases:

- The function call is not supplied with all the arguments from the functions’ argument list.
- The function call has some untagged arguments, e.g. `lm(income ~ prestige, data = newdata)`

![Figure 3.1: Formal arguments of function "lm"](image)

The first case points to the situation where the usage of a function doesn’t always require that all arguments be supplied, as some of the arguments have default values (Figure 3.1). These are still going to be used inside the functions’ source code with their default values, but are most likely going to have a small influence in the behavior of the function.

The second case points to the fact that some function calls have two separate types of arguments: 1-major arguments; 2-context or adjustment arguments. As shown in the example of the second item, where we can see a simple function call of R's linear regression base function. Some arguments are supplied without any names associated (e.g."formula"), and some with their respective names (e.g. "data"). Here the "formula" argument has a decisive influence on the behavior of function "lm", which consequently means that it also has a decisive influence in the time the function is going to take executing. While the "data" variable is obviously important, if we consider it to be a table with 20 columns, the "formula" argument is the one that indicates which of the columns are actually going to be used.

One could argue that the major arguments are static and are simply the ones defined without any default value, but in this work we argue that they vary with the function call semantics. By definition, they are the ones that have the most influence on the behavior of the function and while most of the times the arguments without a default value are indeed the most important ones, this is not always the
case. Thereby we define that a function call’s major arguments are the ones that are not passed with a correspondent name or tag.

Since every operation is handled as a function call, our system’s prediction granularity is along the same lines and considers these major arguments as representative of the function’s execution time. This, along with the combination of arguments that are supplied to a given function call, allows us to generate a new execution time estimate.

In order to do that, we employ a history based approach in which we use previous execution observations of a given function to understand the context and environment of a new invocation, and then predict how much time it will take to execute. More specifically, for each function call observed, we measure how much time it took to execute, measure the size of the object it returned and associate them to a characterization of the arguments it received. The characterization consists in compiling, for every argument supplied, a list of several features previously selected that help define the object within the R interpreter: object name; preview of the object’s value; class; type; number of columns; number of rows and the object size (Figure 3.2). Throughout the rest of the document we will refer to this characterization as “arguments profile”.

Whenever a new prediction request is made, an argument profile of the new function call is created and with the previous entries and their respective profiles, a model of the execution time is built and finally a new prediction is produced.

But for the sake of understanding how all these predictions relate to the final decision we need to take a broader view of the process. A function granularity approach to the execution time requires, naturally, that each function call that composes a given R script is identified, which we do through a syntactic analysis of the code’s call tree. Every time a new function call is found, a new prediction is made and the estimates are added to a total value.

This execution time estimate though cannot be used on its own to fulfill our goal of performing an offloading decision. Therefore we designed a system composed of three major components, which cooperate to provide it with context and meaning. These three components are: the RExecutor, the Market Information and the Incubator, each one operating at different stages with the objective of achieving the fastest decision process possible.

Next we explain with more detail how each of these components work and how they collaborate to the common goal.
3.2 Architecture

The system presented in this work has several components combining efforts to get R users to enjoy the best experience possible while developing software under the R environment. These efforts culminate in the decision of where the program should be executed.

In order to design the architecture of such a system, some requirements were identified:

- **Transparency.** The R user should not need to be aware whether the execution is done locally or remotely.

- **Adaptation.** The system should choose the best computation strategy depending on the available resources.

- **Low Prediction Overhead.** The system must be able to predict the complexity without introducing significant overhead.

- **Precision and Accuracy.** The error associated with the prediction values of the system should be low (accuracy). This should also happen to the prediction variance (precision).

Figure 3.3 shows the system architecture.

3.2.1 R Executor

The R Executor is the module responsible for receiving the input command from the user, starting the whole prediction plus decision process, i.e., responsible for the online mode of the system. In other words, it starts after a user has given the command and finishes after the offloading decision has been made. As seen in Figure 3.3, this module is composed by two inner modules: Complexity Estimator and Decider.

The user provides the R expression whose execution time we want to predict to initiate the process. Because this expression is composed by several other expressions, as a matter of differentiating between the former and the latter, we will call the the former as main expression and all the others simply expressions.

The online modality of the system has an inherent notion of time associated with it since the user is responsible for starting the process. While in this case he doesn't expect to see the return of the process itself (offloading decision), he does expect it to be seamless and to not add too much overhead to the execution. This means that once we commit to initiating the prediction process, we have to be confident that we are capable of estimating the execution time of every function call that exists in the main expression. Otherwise, the prediction would lose much of its strength, since a single unseen function could lead to a misprediction and drastically change the final execution time estimate.

In order to ensure that this situation does not happen, we start by performing a syntactic analysis with the intent of finding all the functions present in the main expression. The analysis consists in inspecting the code's abstract syntax tree (AST), which represents its hierarchical tree structure, and then search for every existent function call.
Figure 3.3: System architecture
This operation can be seen in the beginning of Figure 3.4, which shows the workflow of the live prediction process, followed by the offloading decision. Here, upon receiving a new request from the user, the system uses the syntactic analysis we just mentioned to account for any unseen function. The point here is that, because our prediction is history based, if there is any function of which we have no idea what its behavior is, then we are unable to generate any kind of prediction.

In this regard, whenever a function whose execution history is not found, the RExecutor saves the main expression along with its correspondent environment in a tuple, and immediately proceeds to the local execution instead of proceeding to the prediction. Every time a new tuple is saved, it is stored so that it can be used later on to update the observations’ database.

**Complexity Estimator**

After this first verification, we enter the most fundamental step of the whole system (shown in the grey rectangle), which is responsible for determining the final estimate of the execution time on the local machine.

Having determined that we are going to use execution time predictions of all the function invocations to reach a final value, we need to strip the main expression, and extract these calls so we can then use them. Here we perform a recursive analysis, where the main idea is to recursively advance in the code’s call tree until we find the next function invocation. Every time we find a new one, we predict its execution time and the size of its return object (we explain how later in this section), and add these estimates to a total value.

Although one could note that we already identify which functions exist in the main expression by means of the analysis performed in the beginning, it is important to understand that the syntactic analysis does not take into account the execution flow. But even more important, it ignores the function’s arguments and the environment in which the functions are being called, which is essential for us to be able to create the arguments profile and use it to generate the predictions.

Our technique uses two main ideas to analyse and decompose the main expression:

- Use the control structures (Section 2.4.2) and the assignment operators (<- and <<-) to give context to the analysis and correlate expressions
- Use closure and builtin functions (Section 2.4.2) to determine the final estimate value

While the control structures and the assignment operators do not add, by themselves, any significant value to the global execution time in terms of the time they take to execute, they provide essential context to all the other expressions they are associated with.

When speaking specifically about the control elements, what we mean by saying they provide execution context is that they enable us to understand how each expression is related to each other and apply the correct multiplier to the prediction value. Each of these elements introduces a condition to the execution flow, which influences the number of times a given statement or group of statements is
Figure 3.4: Live workflow
executed. The multiplier is an heuristic value that serves the purpose of estimating the condition and its influence of a given control element.

The assignment expressions, on the other hand, indirectly influence the execution operation with the addition or change of environment’s variables. It is an indirect influence because it is not assured that an assignment operation will have any impact in the outcome of other expressions, since the variable being defined may never be used again. Be that as it may, we consider this impact probability to be high enough to be taken into consideration, specially in a scenario of stacked expressions.

Because the whole point of the prediction is to determine an expression’s execution time without actually executing it, whenever there is an assignment operation in the code, we predict the size of the object that would have been assigned to a certain name. This way, whenever this variable name is used in a future function call prediction during the remainder of the call tree analysis, the system has a way of knowing at least the expected size of the object.

Every time a closure or builtin function call is found, a new prediction is generated. Here we are presented with the challenge of figuring out the call’s behavior while using just the call itself and the environment on which it is supposed to be executed. More specifically, we restrict this behavior to two characteristics: execution time and return object size.

First of all we create a new arguments profile of the fresh function call being predicted. Then we proceed to perform a history lookup, where we identify all the dataset entries correspondent to any function calls that were supplied with the exact same arguments as our new function call. Furthermore, while going through every entry, we identify which are the major arguments of this particular function name.

Different functions have obviously different purposes, but even within the same function, when different combinations of arguments are supplied to a invocation, it hints at an ultimately different goal. By using the selected group of entries mentioned above, we are able to better capture the purpose of the new function call and use similar calls to create a model with better predicting capabilities, since the possibly misleading entries are excluded.

After having not only decided what entries we are going to use, but also which arguments from those entries are going to be part of the model (major arguments), we need to take into consideration the type of the objects associated with those arguments and find out which attribute best represents the complexity of the function call. Whilst the complexity may be related to several aspects of the execution, in this document we refer to the complexity as being directly associated with the execution time, i.e., longer executions correspond to higher complexity values. Here we identified two classes:

- Low significance variables
- High significance variables

The low significance variables are the ones which the object type is a collection (e.g. lists, vectors, matrix, etc), while the high significance variables are the ones which the object is an atomic vector of
type character or numeric. In order to represent the complexity of each of these classes, we use the
object's size for the former and the real object's value for the latter.

The choice of the object size attribute for the low significance variables is based upon the purpose
and context on which the R is inserted. Normally speaking, people use R to perform data analysis and,
being for statistical modeling, data visualisation or data manipulation (to name a few), R has a massive
set of packages available for users to take advantage of. By using 'object size' as a predicting feature,
we are considering that the complexity of a function when supplied with a major argument whose object
is a collection, is related to the object's size. This is expected to happen, specially with this kind of
statistical analysis.

Whenever all the major arguments from the selected entries, along with the ones from the fresh
invocation, happen to be atomic vectors of numeric type, their real number is used as indicator of the
fresh invocation's complexity. Whereas a collection of values has its meaning spread throughout all its
elements, a single numeric object carries all its meaning in its own value, and for that reason we take
special attention to this variant.

Just like a single numeric object, a single character type object contains all the information in its
respective word or sentence. But while in a numeric value scenario, the meaning associated with the
value is likely to present a constant behavior, in a character value scenario, there is no immediate way
to predict what a new word would represent to the function's execution. It would be possible to predict
such situation if further information were to be obtained through a detailed analysis of the function, or
through exaustive observation of previous execution runs, but because we have no possibility for neither
of them, we currently use simply its object size, just like a normal collection. Although this approach
may introduce adicional error in the prediction, it is not expected that data analysis functions receive
character objects as major argument, but rather receive them as optional or complementary ones.

Synthesizing we get the following two situations:

- **Sum of the Objects' Size.** The object size attribute is used whenever there is any major argument
  belonging to the low significance class, or its object is an atomic vector of type character.

- **Sum of the Numeric Values.** A numeric value is used whenever all the major arguments are of
type numeric.

After the history lookup, we get a table like the one from Figure 3.5, where every row corresponds to
a different entry.

Every time a prediction is requested, two different estimate values are computed, representative
of the invocation's behavior: execution time and return object size. In order to reach these estimates
we use the first four columns shown in Figure 3.5 to create the necessary models, which are obtained
through regression analysis.

Here we create two linear models of two dimensions, which are always composed by two sets of
values: 1. dependent variables(y) - ExecutionTime or ReturnObjectSize; 2. independent or predictor
variables(x) - MemorySize or NumericValue. Depending on the arguments' type, we use either the com-
puted MemorySize or NumericValue variables to create the regression functions for both the execution
time and return object size.

If it happens that these models cannot explain with enough certainty the dependent variable variation, we create two additional models where we fit the data to second-order and third-order polynomials. Then we choose the one with the higher Multiple R-Squared metric out of the three. Multiple R-squared is used for evaluating how well a model fits the data. It tells how much of the variance in the dependent variable (the predicted variable) can be explained by the independent variables (the predictor variables). For example, a value of 0.75 implies that the model can explain three-quarters of the variation in the outcome.

![Figure 3.5: "qplot" function | table retrieved after full history lookup](image)

After computing the models, we use the arguments profile from the fresh function call to compute the intended predictions. This means that, depending from the arguments’ type, we use either the computed MemorySize or NumericValue variables to predict not only the invocation’s execution time, but also the object size resultant from the operation.

Whenever a new prediction is finished, the values obtained are summed to the “total” variables, which represent the total execution time and the total memory size to be added to the environment.

**Decider**

This step is responsible for performing the offloading decision based on the information provided by three components:

- **Market Information.** Provides the user’s current balance, the $\text{num\_credits/timeunit}$ and the performance score value ($S$) of each volunteer node.

- **BW Estimator.** Provides the available network bandwidth prediction of each volunteer node.

- **Complexity Estimator.** Provides predictions for both the execution time and the future size of the environment.

After gathering this information, the decider checks if it is worth it to execute remotely by comparing the prediction just made with a predefined offloading threshold. If it is higher than the threshold, then it proceeds to find the volunteer node which offers the best advantages, otherwise, it executes locally.

In order to decide on which machine the program should be executed, the system begins by using equation 3.2 to estimate how much time the main expression is going to take when run on a given remote
machine \( (P_{\text{remote}}) \). We then apply the performance score \( (S) \) to the local execution time estimate \( (P_{\text{local}}) \) we just calculated in the previous step (view workflow of Figure 3.4).

Next, for the sake of finding the most affordable machines out of the available ones, it compares, for each one of them, the user's current balance with the estimated cost (in credits) of an hypothetic remote execution (Equation 3.3).

Then out of those selected machines, the decider picks the one that provide the lowest estimate of the total remote execution time (Equation 3.4). Here we combine the three typical moments that a remote execution is composed of: 1- time to send the environment to the remote machine \( (T_{\text{send}}) \); 2- time to execute at the remote site; 3- time to retrieve the results \( (T_{\text{retrieve}}) \).

\[
P_{\text{local}} = \sum \text{Prediction}(FCall_{\text{non--special}}) \quad \text{(3.1)}
\]

\[
P_{\text{remote}} = P_{\text{local}} \cdot S \quad \text{(3.2)}
\]

\[
\text{Cost} = P_{\text{remote}} \cdot n^0\text{credits/timeunit} \quad \text{(3.3)}
\]

\[
P_{\text{total remote}} = T_{\text{send}} + P_{\text{remote}} + T_{\text{retrieve}} \quad \text{(3.4)}
\]

If no affordable machine is found, then the expression executes locally. Otherwise, the Decider provides the necessary information to the Offloader, which executes the user's input at the selected volunteer node. But before initiating the execution, it saves the main expression and its correspondent environment, just like mentioned at the beginning of the description of Figure 3.4.

Note that the scope of this work does not include the Offloader's operation, which was developed in a previous dissertation [9].

3.2.2 Market Information

This component belongs to the offline mode of the global system, and acts as a bridge between the Offloading Predictor platform and the volunteer platform, which is represented as the Market in Figure 3.3. This external component is assumed to be reliable and secure, so all information provided is accurate.

As we just noted in Section 3.2.1, the Market Information is responsible for providing two major types of information: Volunteers’ Characteristics and Network Bandwidth. In each iteration of the information update process, it begins by requesting the Market for the newest information it has about the volunteers, where it receives two types of information:

- **Volunteer Nodes.** A list of the nodes available, each one with a correspondent list of its characteristics: IP, price in credits per timeunit, performance speedup, Mean Time Between Failures
Balance. Amount of available computing credits of the user.

The Market Information operates in the background of the user’s computer, allowing for the assistance to the online mode to be immediate. It updates the values periodically, so that both the BW Estimator and RExecutor components have the correct information to work properly.

In every iteration, after gathering the necessary information and storing these at the Live Market Info, it starts the Bandwidth Estimator.

**Bandwidth Estimator**

Here we estimate the available bandwidth (BW) for every single volunteer machine in the list of volunteers provided by the market.

The network performance prediction is made continuously, based on periodic measurements, corresponding to each update iteration of the Market Information. This means that the system, with a predefined frequency, actively probes the available volunteers and measures the available network bandwidth. Then, using the historical information gathered, predicts using a Flip-flop filter (Section 2.3) what is the BW in the next time period, until the next measurement. After each volunteer node is handled, their updated values are stored alongside their respective characteristics in the Live Market Info data structure.

By doing this cyclic process in the background we can ensure that whenever the RExecutor requests information, it is available right away.

**3.2.3 Incubator**

The quality of a predictive system with a heavy reliance on a dataset of previous observations, directly depends on how good and representative of the situation that dataset is. Because of that we need to take special attention to how we populate our dataset for it to be representative enough.

Generally speaking, a dataset becomes better, i.e, with better prediction capabilities, as new entries are added. However, using just the amount of entries to judge the quality of a dataset may be misleading, since it may be very good at predicting a certain type of event, but the moment there is a certain change in the context, it is no longer able to predict the outcome. This way we need to have not only lots of observations, but observations that are representative enough of the context we are handling with.

In this work, the context is a function call. When speaking about function calls, the possible variants of the context are not only the different possible values that the function’s arguments may take, but also the different possible combinations of given arguments. For example, when considering function $f(a, b=5)$, we need to be aware that $f$ will perform differently if its argument $a$ is a large vector, a small vector or not even a vector. Moreover, we need to consider $f$’s behavior when it receives both $a$ and $b$, or just $a$.

So in an attempt to populate our dataset with observations representative enough of the function’s purpose and utility, we identified two types of situations where we can use real examples, instead of artificially made ones:
– Use official examples

– Use real user examples - dynamic learning

The first one makes use of the official examples from the CRAN repository. Taking advantage of the fact that each function that belongs to a package from the CRAN repository has a correspondent usage example script, we identify the function calls from the function being populated and use the technique described in Section 3.1 to generate new argument profiles for every function call. Afterwards, we execute the calls themselves with the arguments given within the correct environment. We measure their correspondent execution time and size of the returned object and add them to the profile. These three elements together make the structure of a new dataset entry.

The second one uses most of the principles we just described, but uses user generated scripts instead of example scripts. As mentioned in Section 3.2.1, every time a new request is made by the user, a new tuple containing a script in R and its respective execution environment are stored. These are used by the Incubator and, just like for the official examples, we identify all the function calls in the script. This means that instead of looking for one function name in particular like we did when inspecting an example script of a given function, we use every single identified call to generate another entry for their correspondent function’s dataset. This solution is not only able to populate the datasets for the functions which could not be populated through their examples, but also improves the prediction capabilities of the current datasets as new entries are added to their total amount.

As a matter of differentiating functions with the same name but from different packages, each function is identified by its name and its namespace. Which leads to a specific type of functions functions, that are identified not based on their syntactic context but based on where and when they were created.

Like in any programming language, R allows users to create any additional functions they find necessary. These functions do not belong to any predetermined package. Instead they are stored in the execution environment. And although they can be used like any regular function, with its own body and arguments, they are only defined upon the execution of their corresponding instructions by the R interpreter.

User functions are a special case because there is no way to populate these functions beforehand, but rather they have to be populated as the function is going to be executed through the regular use of R. Besides that there is the fact that this type of functions are inevitably susceptible of having its source code changed at any time, and consequently its purpose and complexity. Unless there is some way of determining its impact, such change would mean that every previous observation becomes obsolete and can no longer be used.

We handle this uncertainty by storing the previous observations under an identifier based on the function’s name, the environment where the function was defined and the source code. Any time there is a change either in the arguments list or in the source code, that change will reveal itself in the identifier and the system will interpret it as a new function.

The Incubator operates in full offline mode, ensuring that the populating procedure does not increase
the system's prediction overhead.
Chapter 4

Implementation

This chapter will describe the implementation of the Predictor System.

The system (Figure 3.3) was developed under two different programming languages: R and Python. R was used for every module that exclusively operates in local fashion, i.e., that does not need to communicate any of the external components (RExecutor and Incubator). The Market Information, because it requires not only to communicate with the Market, but also with the different Volunteers, was implemented under Python.

The point of dividing the architecture’s implementation in such manner was to take advantage of Python’s optimized libraries for external communication, while maintaining a seamless integration with how the user operates with R itself, which is through its interpreter. In order to achieve this integration, we’ve decided to implement the system’s interface like a regular function, allowing the user to take advantage of our system as if he were calling any function. After the invocation, where the R expression is supplied, the decision operation is done through scripts that automate the process.

Actually, none of the three main components that were described in the previous chapter need continuous user interaction. They just need for its process to be jumpstarted whenever necessary. The local machine will have the Market Information process running in the background, and the RExecutor running in the R interpreter process, like a normal function invocation would. If the user Pretends to use the Incubator at the same time of the decision operation, then it runs in a separate R process, never being aware that the other is currently running.

Next we present a detailed description of how the main techniques used in our Predictor System were implemented and took advantage of R’s internal representation, finishing of the chapter with an explanation of how the RExecution communicates with the Market Information, and how the latter communicates with the external components.

4.1 Argument Miner

The core of our ability to predict the execution time of a certain script in R is the way we profile the function calls through their arguments. Whether it is the Incubator (Section 4.3), using this mining
technique to generate new entries, or the Complexity Estimator (Section 3.2.1) component to create a fresh profile while computing a new prediction, the arguments profile is of the utmost importance for the proper functioning of the system.

While the two variants use basically the same concepts, they differ in a fundamental aspect which is the time they have to characterize the arguments. In the remainder of this section we will describe how we generate the argument profiles in an offline mode, and then detail what differs when the profile is made in a live prediction context.

Upon receiving a new function call, we decompose it so that its function name and arguments can be identified. By decomposing it we take advantage of the fact that R stores function calls as a list, where the first element is the function name and the rest of the elements correspond to the arguments.

Depending on whether the supplied arguments are tagged or not, the list of elements can be named, corresponding to a tagged matching of formal and actual arguments. This allows us to easily parse the different elements that compose a function call and manipulate them as they are analysed, enabling an immediate identification of which are the major arguments. All the arguments that were supplied without a corresponding tag are then identified through positional matching (Section 2.4.2), where they are bounded to any unmatched formal arguments, in order.

Using the arguments list, we use a “for” loop to iterate through every argument and characterize it. Each argument is characterized using the following attributes:

- **Name.** The variable name. Displays "" if no name was associated.
- **Value.** The object's value. If the object is a collection then it has stores the value as NULL.
- **Class.**
- **Type.**
- **Length.**
- **Number of Columns.**
- **Number of Rows.**
- **Object Size.** The memory size which the object occupies.

These are the eight fundamental attributes that we look for when characterizing a basic object, which is essentially a non-expression argument. But whenever an expression is passed as an argument, we need to look deeper and determine how we are supposed to handle it.

In R, any argument of type "language" is of class "call" by default, which leads us back to how in R any call is a function call (Section 2.4). Being a function call, we have to be aware that functions in R are divided into three different types: Special, Built-in and Closure (Section 2.4.2), handling their arguments differently.

In light of this, we have decided to separate them by how they handle their own arguments:
- **The function evaluates its arguments - Closure and Builtin.** Every time a closure or builtin function is passed as argument, we simply execute them so we can characterize their return object like we do for the basic objects.

- **The function does not evaluate its arguments - Special.** Since special functions do not evaluate their arguments, we characterize each of them individually and then consider the sum of their "objectSize" values as the total object size (Figure 4.1). The sum value is stored in a new attribute representing the size of the objects used in the language operation (*sizeObjectsUsed*).

After every argument is properly characterized, we finish with a named "list of lists", where each element of the main list corresponds to the arguments supplied to the function call (Figure 4.1).

![Figure 4.1: Arguments profile example for function call: lm(weight ~ group, method = "model.frame")](image)

When comparing this description with how it would do when put in a live prediction context, the glaring problem is the execution of closure and builtin functions, considering how it would represent a huge prediction overhead. In this online context, we have to make use of what objects are already on the environment, since any execution would incur in too much added time to what we can afford.

For these reasons, whenever closure or builtin functions are supplied as arguments to the function call we are trying to predict, we purely use other predictions to estimate the characteristics of the return object. More specifically, we estimate their memory size and use it as a defining attribute of the object, i.e., the characterization of the arguments’ attributes consists just on its memory size, which is the attribute needed for the prediction (Figure 4.2).

Figure 4.2 shows an example of an arguments profile of function "crossprod" where function "Rnorm"
is provided as argument (closure). Here we can see that the prediction of the return object’s size was successfully done, and the estimate value is assigned to the `sizeObjectsUsed` attribute.

4.2 Call tree analysis

This technique is fundamental in allowing us to handle multiple expressions at once, i.e., in making available to the user the option of suppling multiple expressions to our system instead of one expression at a time. This provides the user with a more intuitive interaction, although at the expense of a degree of prediction accuracy. Such accuracy is affected not only by the heuristics this technique uses, but also by the fact that some predictions have to be made using other predictions’ results.

This technique begins by converting the main expression, which is supplied by the user, into a list. From this conversion we obtain a list where each element corresponds to the different independent expressions of the given code. By independent we mean that the expression is at the top of the call tree.

In Figure 4.3 we can see that although the `for` loop uses variable “z”, they are at the same level in the call tree, which we classify as independent.
After the conversion, we initiate a sequential analysis for each of the list's elements (inner expressions). Here the analysis of each element could be made in parallel fashion, but then we would miss on the possibility of using any information from a previous prediction into a new prediction. This would mean that we wouldn't be able to successfully predict expressions that use variables assigned by any previous ones, since we would have no idea of what that variable is.

Another alternative would be to perform an individualized offloading decision for every independent expression. In other words, we could assume every single independent expression represents a new invocation to our system, triggering its execution every time a decision was made. This would surely diminish the prediction error as we would be dealing with the real values, but this operation would incur in too much prediction overhead resulting from the repeated environment transmission to the remote machines.

Taking this into account, we have decided to implement a sequential analysis, where each of the expression's analysis returns a prediction value of how much time (in seconds) the correspondent expression's execution is expected to take. After reaching a final value, the total prediction estimation is returned, and the offloading decision takes place.

In Section 3.2.1 we saw that the main concept we utilize to maintain the execution context, even without executing the code itself, is to utilize the language flow control elements.

As seen in (Section 2.4), and again in the previous section, R stores every elementary R expression in a function call form, i.e, in a list where the first element is the function name and the rest of the elements corresponds to the arguments. This is also valid to the control structures and the assignment operators, as these language elements are simply viewed as another expression with their own set of arguments. Moreover, a compound expression, which is essentially a collection of expressions grouped together, is stored just like any other expression where the function name is "\{".

Our recursive analysis makes use of this constant internal representation to decompose each expression in its function name and arguments.

We use the function name to differentiate between the control elements:

- **For.** : "for"(var, list, expr)
  Whenever a "for" loop is encountered, we measure how many elements the "list" has and use this value as a heuristic factor for the added complexity which the loop will represent to the execution.

  Then we call the same call tree analysis technique and supply it with "exp" (read: expression), which corresponds to the code the loop iterates on. The execution time estimate that results from this inner analysis is then multiplied by the heuristic factor we previously determined.

- **If.** : "if"(cond, expr)
  Whenever a single "if" statement is encountered, we run our recursive analysis on its correspondent "expr" element. The execution time estimate that results from this inner analysis is then multiplied by a heuristic factor of 0.5.
The point here is that, without executing the condition to determine whether the result is true or false, we are left with the essence of what a binary true/false decision is, which is a 50% probability of executing and other 50% of not executing.

Because the non-executing option corresponds to zero seconds, here we just apply the 50% probability to the time estimate.

- **If...Else**. : "if"(cond, expr1, expr2)

In R there is no "If...ElseIf...Else" statement. Instead it uses chained "If...Else" statements. For this reason we describe both cases in the same topic.

This case follows the same principles which a simple "if" statements does, heuristically speaking. In order to determine the execution time of a chained "If...Else" statement we individually analyse each of the expressions that make up the sequence of conditions and apply an heuristic factor correspondent to the ratio between one and the number of execution possibilities within the chain.

So, taking as example an "If...ElseIf...Else" chain, with their respective expressions being "expr1", "expr2" and "expr3", the prediction time would be: \[
\frac{1}{3} P_t(expr1) + \frac{1}{3} P_t(expr2) + \frac{1}{3} P_t(expr3).\]

- **While**. : "while"(cond, expr)

The "while" control flow element, unlike "for", does not provide any hints about how much time the execution is going to take. Here there is no way of knowing beforehand when the loop will stop, since it is necessary to execute the code to know if the stop condition is true or false.

In fact, the only guarantee we have is that its source body is going to be executed at least once. Because of this, we consider that the heuristic factor is equal to one.

- **[, [ , $. : "index construct"(object, tag)

Whenever any of these three R's index constructions are encountered, we simply ignore the tag and run the same recursive analysis on the "object" element.

We further analyse the object because it can formally be any valid expression, as long as it denotes or evaluates to a subsettable object.

- **<- , <<- : "assignment symbol"(variable name, object or expr)

An assignment operation, unlike the elements above, has a direct impact in the execution environment. For this reason we separate how we handle them in two, accounting for the situations where the third element of the internal representation is either an object or another expression.

Because the value of an assignment operation always surpasses by far the negligible amount of time it takes to execute, every time there is an object assignment, the expression is executed and the environment is updated like a it normally would. The fact that the object already exists in the environment means R won't have to compute it. Instead it will exclusively perform an assignment operation.

On the other hand, if the right side of the assignment corresponds to an expression, we use the call tree analysis like before, but this time to predict the size of the return object. This size estimate
is then assigned to the "variable name" inside a local environment object, meaning that the return object will be represented by an estimate of its memory size.

Every time that none of these "function names" match the one from a fresh expression, we trigger the prediction of both the execution time and the returnObjectSize for that function call.

4.3 Incubator

This technique, which gives the name to the component from our architecture (Figure 3.3) that uses it, does exactly what one would expect. When given a function call and its correct execution environment, it transforms these two raw pieces of data into a compiled characterization of that function call (Figure 4.4 in the next page), which we save as a dataset entry.

We use a combination of the two techniques we just presented in Sections 4.1 and 4.2. We start by identifying the function calls existent in the R script using a modified version of the call tree analysis technique. Here the difference is that all the code related to the heuristics and to the prediction operation in general was removed, leaving just the code that allows us to advance the call tree until we find the next function call. Besides that, every expression is executed as we advance in the tree and they are no longer needed, allowing for the environment to be always updated.

Upon finding a function call, instead of generating a new prediction like we do when considering the online context, we use the Argument Miner (Section 4.1) to generate a new arguments profile. After that, as seen in Section 4.3, the function call is executed and its correspondent execution time and size of the returned object are measured and saved. These three elements, along with the function call itself, are then aggregated in a named list, making the structure of a new dataset entry.

Figure 4.4 shows a structure example of a dataset entry, where we can distinctively see the four elements that make up the list. Notice that instead of having another indexing level and storing the arguments profile there, as a matter of simplifying the access to each of the different arguments, we have decided to leave them in the same indexing level as the other descriptive elements (last three).

After constructing the entry, it is added to the correspondent dataset. The dataset consists of a list, where the first element contains a vector with all the formal arguments of the function that were previously supplied with no tag, and the remainder of the elements correspond to the different entries. We also refer to the first element as the major arguments list, and it is updated whenever a new entry is added to the dataset.

Because we need to manipulate the different entries, we serialize this list using the R’s native function (saveRDS) and save it under an identifier, composed by the function name and the namespace.
List of 7
$data
..$ symbolName: chr "mtcars"
..$ class : chr "data.frame"
..$ typeof : chr "list"
..$ length : int 11
..$ NCOL : int 11
..$ NROW : int 32
..$ objectSize: num 6736
<size>
..$ symbolName: chr "cyl"
..$ class : chr "numeric"
..$ typeof : chr "double"
..$ length : int 32
..$ NCOL : int 1
..$ NROW : int 32
..$ objectSize: num 296
$x
..$ symbolName: chr "mpg"
..$ class : chr "numeric"
..$ typeof : chr "double"
..$ length : int 32
..$ NCOL : int 1
..$ NROW : int 32
..$ objectSize: num 296
$y
..$ symbolName: chr "wt"
..$ class : chr "numeric"
..$ typeof : chr "double"
..$ length : int 32
..$ NCOL : int 1
..$ NROW : int 32
..$ objectSize: num 296
$ExecutionTime : num 0.00128
$returnObjectSize: num 11008
$Command : chr "qplot(mpg,wt,data=mtcars, size=cyl)"

Figure 4.4: Example of a dataset entry's structure
4.4 History Lookup

As we mentioned in Section 3.2.1, our prediction generation process starts up with a history lookup, where we identify, for a given function call, what are the dataset entries that have the exact same arguments. While at it, we use the major arguments to compute the entries’ complexity value, which later on are given to the regression function. In the remainder of the section we refer to the function call we are trying to predict as “fresh call”.

We begin by loading the correspondent dataset into the environment, which had been serialized upon saving. Next, we read the major arguments from the dataset’s first element and proceed to iterate through all the entries.

For each entry there are four things that we need to discover:

1. Does the entry correspond to a function call with the same arguments as the fresh call?
2. What are the arguments that are going to be used to represent the entries’ complexity?
3. How should we represent the complexity of this particular function (memory size vs numeric value)?
4. What is the actual value that represents the entries’s complexity?

Every new iteration, we begin by comparing the entries’ arguments with the ones from the fresh call. If they aren’t an exact match, then we immediately jump to the next entry in the dataset, otherwise we proceed to further analysis.

At this point we decide on what are the arguments we should use to represent the entries’ complexity. The perfect situation would be to always use the major arguments, but although not likely, it may happen that all the previous observations correspond to function calls whose arguments were all tagged, meaning that no major arguments were identified. For this reason we check if the major arguments list we read before starting the iteration is not empty, and if it concurs, we use the mentioned major arguments, if not, we use all the entrie’s arguments.

Standing inside the entry, and having decided which of its arguments we are going to use, we need to understand if we’re dealing with a funcion which complexity varies with a numeric value, or if it is the memory size of its arguments that are the deciding factor. In order to do that, and before iterating through each of these arguments, we define a boolean variable indicating whether we should use a numeric value or not (default is “true”). Every time we encounter an argument whose type is not numeric, or is in fact numeric but it’s length is higher than one, we change this boolean variable to “false”.

As we go through these arguments’ profiles we sum their “objectSize” attribute to the entries’ total. In the cases where the argument is an atomic vector of numeric type, i.e, a number, we sum this value to the total “numeric value” of the entry.

Before moving to the next entry in the dataset, the last thing we do, is to add a new row to a table containing all the information we were able to obtain related to this entry.
<table>
<thead>
<tr>
<th>MemorySize</th>
<th>NumericValue</th>
<th>ExecTime</th>
<th>ReturnObjSize</th>
<th>ArgsUsed</th>
<th>ClassOfUsedArgs</th>
<th>ArgsPassed</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.58</td>
<td>NA</td>
<td>0.012631</td>
<td>10.45</td>
<td>x / y</td>
<td>numeric_vec / numeric_vec</td>
<td>data;x;y</td>
<td>qplot(mpg, wt, data = mtcars)</td>
</tr>
<tr>
<td>0.58</td>
<td>NA</td>
<td>0.011284</td>
<td>10.45</td>
<td>x / y</td>
<td>numeric_vec / numeric_vec</td>
<td>data; facets;x;y</td>
<td>qplot(mpg, wt, data = mtcars, facets = a_1)</td>
</tr>
<tr>
<td>0.29</td>
<td>NA</td>
<td>0.003854</td>
<td>10.33</td>
<td>x</td>
<td>numeric_vec</td>
<td>data;x</td>
<td>qplot(mpg, data = mtcars)</td>
</tr>
<tr>
<td>0.29</td>
<td>NA</td>
<td>0.001864</td>
<td>10.33</td>
<td>x</td>
<td>numeric_vec</td>
<td>data;x</td>
<td>qplot(gear, data = mtcars)</td>
</tr>
<tr>
<td>0.58</td>
<td>NA</td>
<td>0.001550</td>
<td>10.45</td>
<td>x / y</td>
<td>numeric_vec / numeric_vec</td>
<td>data;x;y</td>
<td>qplot(mpg, carb, data = mtcars)</td>
</tr>
<tr>
<td>0.58</td>
<td>NA</td>
<td>0.001498</td>
<td>10.75</td>
<td>x / y</td>
<td>numeric_vec / numeric_vec</td>
<td>data;x;y</td>
<td>qplot(mpg, wt, data = mtcars, colour = cyl)</td>
</tr>
<tr>
<td>0.58</td>
<td>NA</td>
<td>0.001335</td>
<td>10.45</td>
<td>x / y</td>
<td>numeric_vec / numeric_vec</td>
<td>data;x;y</td>
<td>qplot(drat, qsec, data = mtcars)</td>
</tr>
<tr>
<td>0.58</td>
<td>NA</td>
<td>0.001329</td>
<td>10.75</td>
<td>x / y</td>
<td>numeric_vec / numeric_vec</td>
<td>data;x;y</td>
<td>qplot(mpg, wt, data = mtcars, size = cyl)</td>
</tr>
</tbody>
</table>

Figure 4.5: Example of a full history lookup of "qplot" function from package "ggplot2"

<table>
<thead>
<tr>
<th>MemorySize</th>
<th>NumericValue</th>
<th>ExecTime</th>
<th>ReturnObjSize</th>
<th>ArgsUsed</th>
<th>ClassOfUsedArgs</th>
<th>ArgsPassed</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.58</td>
<td>NA</td>
<td>0.012631</td>
<td>10.45</td>
<td>x / y</td>
<td>numeric_vec / numeric_vec</td>
<td>data;x;y</td>
<td>qplot(mpg, wt, data = mtcars)</td>
</tr>
<tr>
<td>0.58</td>
<td>NA</td>
<td>0.001550</td>
<td>10.45</td>
<td>x / y</td>
<td>numeric_vec / numeric_vec</td>
<td>data;x;y</td>
<td>qplot(mpg, carb, data = mtcars)</td>
</tr>
<tr>
<td>0.58</td>
<td>NA</td>
<td>0.001335</td>
<td>10.45</td>
<td>x / y</td>
<td>numeric_vec / numeric_vec</td>
<td>data;x;y</td>
<td>qplot(drat, qsec, data = mtcars)</td>
</tr>
</tbody>
</table>

Figure 4.6: Example of a history lookup with argument matching using: `qplot(income, education, data=Prestige)`
In Figures 4.5 and 4.6 we respectively show examples of what the final tables would look like if we used all the entries in the dataset and if we used only the arguments that match the function’s call supplied arguments.

The total history lookup finishes when all the entries are iterated, at which point the table is ready to be used to generate the predictions. Along with the table, the variable indicating if the function is numeric or not, is also returned. This is then used to decide whether the MemorySize or the NumericValue columns should be used.

### 4.5 Component Communication

While most part of the communication between the components in our architecture (Figure 3.3) is done through simple invocations (all within their own process), there are three that are more complex:

1. RExecutor $\leftrightarrow$ Market Information
2. Market Information $\leftrightarrow$ Market
3. Market Information $\leftrightarrow$ Volunteers

In a closer look at these three items we can see that there is a common element between the three: the Market Information. This is essentially the component that acts as an intermediary between the local system and the external components, acting in a three-way front.

The Market Information, as we saw in Section 3.2.2, has two main responsibilities within our system which consist in being always available to the RExecutor and periodically updating the external information, being from the Market or from the Volunteers.

Having this in mind, we implemented this component so that it consists in a Python process with two threads, each one corresponding to the two sides of the figurative bridge between the inner system and the external side. In other words, one thread remains listening for any information requests from the RExecutor and the other operates in a periodic loop, updating the information periodically.

The inner part of the communication required more attention since we needed to communicate between an R process (RExecutor) and a Python one (Market Information). This interprocess communication is done through regular sockets, but because we needed an interface to convert the Python objects into R objects, whenever the RExecutor requests the latest information to the Market Information, it does so using the rPython package [55], which allows Python code to be called from R. By performing the request in a Python script we are able to do the communication purely in Python and then get a seamless conversion to R from the rPython interface.

The second item has to do with the communication with the Market component. We implemented this as being a regular socket communication, where the Market would have a certain IP address and would be listening at a certain known port for requests.

Finally, whenever we measure the network’s quality for each of the Volunteers, we obviously need to communicate with them. Here we use the latest version of iPerf [56] - iPerf3 - which is a tool for active
measurements of the maximum achievable bandwidth on IP networks. The volunteers just needs to have an iperf server running, that our system by running an iperf client with the remote host's IP address and port, we are able to get a diagnostic object of the network bandwidth between the two machines.

Because the measurement is done for every available volunteer, every time the list of available ones is updated (after the communication with the Market), we create a pool of workers, which we call "scouts". Each of these scouts runs an iperf client in a subprocess and is responsible of measuring the available bandwidth of the specific volunteer it is assigned to. Whenever a scout finishes, its respective iperf client is stopped.

When all the scouts have finished, the information is saved in the Like Market Info and the Market Information enters another still period until the next update iteration.
Chapter 5

Evaluation

The strength of our system ultimately depends on the weight of which the prediction process has on the
global execution (prediction overhead) and on the quality of the offloading decision. By quality we mean
the capacity of the system to determine if the execution should be done locally or remotely.

In order to validate these two factors, two scenarios were considered:

1. R script with multiple instructions
2. R script with one instruction

The first scenario represents the situation where the user provides multiple instructions to our system,
which may or may not be directly related to each other through control commands, or by depending on
their respective outcomes. The second one represents a simple situation where the user pretends to
use our system to execute a single instruction, depending only on the current environment.

These were selected in an attempt to represent two opposite execution situations with inherently
different challenges. While the former presents the challenge of predicting the instructions’ behavior
within a higher error probability environment, the latter stresses the ability to predict without incurring a
heavy load in the form of prediction overhead.

Because this is just a prototype of the global system, the evaluation environment consists in a single
machine, where the behavior of the Market and the different Volunteers nodes is simulated. For the
purpose of testing our system, the information that would be obtained from the communication with these
components is predefined Table 5.1). The evaluation is performed under a Linux OS (Elementary), with
a 16Gb RAM, and core i7 processor, while running the 3.4.1 version of R.

Since here we are just interested in determining the impact of the prediction error on the decision,
we assume that the local user has enough credits for all the considered volunteers. This means that,

<table>
<thead>
<tr>
<th>User</th>
<th>Available Bandwidth</th>
<th>Performance Score</th>
<th>Credits/Unit</th>
<th>Balance in credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volunteer 1</td>
<td>1Mb/s</td>
<td>0.5</td>
<td>•</td>
<td>-</td>
</tr>
<tr>
<td>Volunteer 2</td>
<td>5Mb/s</td>
<td>0.75</td>
<td>•</td>
<td>-</td>
</tr>
<tr>
<td>Volunteer 3</td>
<td>10Mb/s</td>
<td>2</td>
<td>•</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.1: External components’s information
for these tests, we use a decision operation purely based on the estimated execution times that each machine provide, ignoring the impact the credits may have on the selection.

The datasets for the functions belonging to R’s core packages were populated using the first technique described in Section 4.3, which makes use of real examples from the CRAN repository. Moreover, the functions which appear in the R script used for the evaluation were also populated using two similar calls. These were profiled with different argument values, while maintaining the same call syntax.

In order to evaluate our system we used the Prediction Overhead Percentage, the Symmetric Mean Absolute Percentage Error (SMAPE), the Mean Absolute Deviation (MAD) and then measured the impact they have on the decision accuracy. We use the first metric as an indicator of the weight our system has on the execution time. We use the first metric as an indicator of the weight our system has on the execution time. The following two metrics are two of the most renown forecast-error metrics [57]. We use MAD to measure our system’s forecast error when scale-dependent, and SMAPE when scale independent (in percentage terms). Notice that we do not use the more common MAPE metric because, when dealing with low volume items, it can give a distorted picture of error. Finally, we measure the impact of our system’s forecasting error in the final decision’s accuracy.

In the next sections we evaluate, for both scenarios, the metrics we have just mentioned. We then proceed to an overview analysis over the results of the evaluation.

\[ SMAPE = \frac{2}{N} \sum_{k=1}^{N} \frac{|F_k - A_k|}{F_k + A_k} \]  

\[ MAD = \frac{1}{N} \sum_{k=1}^{N} |F_k - A_k| \]

### 5.1 Multiple Instructions

As a means to carry out the evaluation with a real life usage of the R’s functions and capabilities, we used the R-Benchmark 25 script [58], which is globally used in the R community as an utility to measure the R’s performance under different machines. This script benchmarks numerical operations such as loops and matrix operations and is comprised by three separate benchmarks: I. Matrix Calculation; II. Matrix Functions; III. Programming.

In this scenario of having multiple instructions possibly depending from the execution outcome of previous ones, we evaluated four examples separately:

1. Full R-Benchmark 25 script
2. Benchmark I : Matrix Calculation
3. Benchmark II : Matrix Functions
4. Benchmark III : Programming

After every operation in each of the three benchmarks, the script uses `remove` calls to clean up the environment. This would not only mean that the environment’s size before their execution would be exactly the same, but also mean that the environment after the execution would remain unchanged.
In an effort to determine the impact of the prediction error in the decision we have removed these clean up calls so that the initial environment’s size may differ from one another and different values are handled by our system.

5.1.1 Decision Accuracy

Here we start by comparing our prediction values ($P_{Local}$) with the real elapsed time, which is shown in Figure 5.1. This chart enables us to have a global view of the error associated with the forecasts without forgetting the time scale correspondent to each of the execution runs.

Using the measured data we obtained the following MAD (equation 5.2) : 8.02 seconds

![Figure 5.1: Execution estimate vs observed time](image)

But from an evaluation standpoint we also need to be able to compare these different execution runs using a scale independent metric, which is why we use SMAPE to characterize the prediction error of our system. Measuring the size of the error in percentage terms allows an easier interpretation of the forecasting accuracy, where most of the times it is not necessary to consider the context on which the values were obtained in order to have a correct understanding of the situation. In the chart of Figure 5.2 we can see the absolute percent error correspondent to each of the four examples.

Using these values, we obtained a SMAPE (5.1) of 13.39%, which means that our forecast for multiple instructions scenarios has an average prediction deviation of 13.39% from the correct value.

Next we introduce the external components into the context and evaluate the system’s decision accuracy. For the purpose of testing we have defined the offloading threshold to be 60 seconds, which
means that if the execution time prediction is lower than 60 seconds the local execution is promptly initiated.

In order to determine if the prediction error is critical to the offloading decision, we need to simulate a decision using our predicted values and then compare the decisions we get to the ones we would make if we used the actual observed values.

In Figure 5.3 we show the main values for a prediction situation and in Figure 5.4 we show the real observed values, both containing the respective decision of all four cases in study.

<table>
<thead>
<tr>
<th></th>
<th>Initial Env Size</th>
<th>Exec Time</th>
<th>Final Env Size</th>
<th>Total Exec Time (s)</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Full R-Benchmark</strong></td>
<td>4 Mb</td>
<td>168.77 sec</td>
<td>15.78 Mb</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Volunteer 1</td>
<td>104.17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Volunteer 2</td>
<td>130.53</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Volunteer 3</td>
<td>339.52</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>I.</strong></td>
<td>4 Mb</td>
<td>73.89 sec</td>
<td>37.397 Mb</td>
<td></td>
<td>Volunteer 1</td>
</tr>
<tr>
<td></td>
<td>Volunteer 1</td>
<td>78.34</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Volunteer 2</td>
<td>63.69</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Volunteer 3</td>
<td>151.91</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>II.</strong></td>
<td>34.8 Mb</td>
<td>86.49 sec</td>
<td>76.72 Mb</td>
<td></td>
<td>Volunteer 2</td>
</tr>
<tr>
<td></td>
<td>Volunteer 1</td>
<td>154.76</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Volunteer 2</td>
<td>87.17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Volunteer 3</td>
<td>184.13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>III.</strong></td>
<td>62.8 Mb</td>
<td>8.37 sec</td>
<td>71.39 Mb</td>
<td></td>
<td>Local</td>
</tr>
<tr>
<td></td>
<td>Volunteer 1</td>
<td>8.37</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Volunteer 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Volunteer 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As mentioned in Chapter 3, we use the total execution time (Equation 3.4) values to determine which machine we expect to return faster results. This means we take into consideration not only the execution time itself, but also the time it takes to send the environment to the remote location and then return the
Both tables are divided into 2 groups. In the first one we have, for each example of the scenario, the values which are going to be combined with the available bandwidth and the performance score from table 3.4. On the second one, we present the calculated values for each of the possible execution nodes and the correspondent offloading decision.

Here we can see that overall, the prediction error did not critically impact the decision quality as it was correct three out of four times (75% accuracy). What did impact the decision were our predictions of the final environment’s size which overestimate the real value. When looking to example II, which was the one that was incorrectly decided, we see that the additional 14 Mb we estimate the final environment to have, corresponded to a deviation of approximately four seconds which in this case was enough to cause a wrong decision.

Also, we can see that for example III the local execution time value is less than the defined offloading threshold, and being so it is executed locally without even considering the remote volunteers.

### 5.1.2 Prediction Overhead

Here we find out how heavy our system is (in percentage terms) relative to the local execution time, i.e., how much longer does the code take to execute if our system is used and the decision is to execute locally.

In the chart of figure 5.5 we can see the real elapsed time of the examples side by side with their respective total overhead value in seconds. These values translate in percentage terms to the chart of figure 5.6.

In the second chart we show these percentages on top of one another, enabling a quick interpretation of the relation between the overhead generated with the real elapsed time. By analysing the different columns we can easily understand that, for example the full R-Benchmark’s prediction adds an overhead of just over 5% to the total execution.

The mean overhead percent of the four examples evaluated results in a 16.2% increase of the local execution time. Here is important to note that, as we can see in the chart 5.6, the geometric mean is
greatly influenced by the example III., where the prediction overhead stays at 52% (these values are further discussed in 5.3).

![Figure 5.5: Prediction overhead - time](image)

### 5.2 Single Instruction

In this scenario we decided to evaluate the performance of our system when dealing with a single function invocation, which means that the unpredictability of an instruction sequence previously present is now removed from the equation. Here we used four distinct invocation cases:

1. `crossprod(a)`
2. `crossprod(b)`
3. `c(a)`
4. `c(b)`

Where `a` is a random 2800x2800 matrix and `b` is a random 100x100 matrix.

These cases were selected in an attempt to evaluate how primitive (`c`) and non-primitive (`crossprod`) invocations influence our system’s performance when given a big (matrix `a`) or a small (matrix `b`) argument.

The first two use the R’s crossproduct function, which performs a cross-product between the two arguments it receives. If receives only a single argument, then it assumes the second argument as being the same as the first one.

The last two, use the R’s native vector creation function, which in these two cases creates single element vectors with matrix `a` or matrix `b`, respectively.
5.2.1 Decision Accuracy

Following the same methodology as explained in Section 5.1.1, we started by calculating the MAD using the data shown in Figure 5.7. After a quick chart observation we can easily understand that the error is considerably smaller when compared to the one obtained in the scenario previously evaluated.

When calculating the mean deviation we obtained a value of approximately 0.3 seconds.

Next we translated the measured information in time units to percent terms, and constructed the chart of figure 5.8. Here we can see that although the crospross(a) invocation shows a small prediction error percentage wise, the other three invocations show prediction errors considerably bigger than every other measurement previously performed. When we calculated the SMAPE, the obtained prediction error was of 130%, which follows what was noted when analysing the chart. In spite of achieving a good prediction error (2.5%) when performing the cross-product with matrix a, the influence it has on the computed average is overwelmed by the rest of the invocations, resulting in an incredibly high error percentage.

When evaluating the impact of the 0.3 seconds of prediction error in the offloading decision, the obtained decision accuracy was of 100%. For this scenario we can easily see that the MAD present in our estimations do not impact whatsoever the decision, specially since the offloading threshold defined for these tests was 60 seconds and all the measurements were well below this value. In all four cases the decision would be to promptly execute the code on the local R interpreter.
Figure 5.7: Execution estimate vs observed time

Figure 5.8: Absolute percent error


### 5.2.2 Prediction Overhead

When measuring the prediction overhead percentage of the four different invocations we can observe a similar occurrence of what we just noted when measuring the prediction accuracy. When comparing the overhead generated by the first invocation with the others under the same chart, like we do in figure 5.9, the overhead of the first column seems close to zero, when it is actually of approximately 5%. These 5% follow what we saw for the overhead measurements in the previous scenario, but the rest of the values are significantly bigger than any we’ve had measured yet.

![Figure 5.9: Prediction overhead percentage](image)

These measurements result in a mean prediction overhead of 1185%, which would mean that using our system, increases, on average, 1185% of the local execution time. This value, while meaningful, does not totally represent the system’s behavior since the measurements present very high variance. It is necessary further consideration, in order to get a proper appreciation of the system’s overhead when dealing with single instructions, which we do in the following section.

### 5.3 Results Validation

In table 5.2 we present a synthesis of the measured values which characterize our system.

<table>
<thead>
<tr>
<th></th>
<th>MAD</th>
<th>SMAPE</th>
<th>Prediction Overhead Percentage</th>
<th>Decision Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi Instructions</td>
<td>8 s</td>
<td>13.4 %</td>
<td>16.2 %</td>
<td>75 %</td>
</tr>
<tr>
<td>Single Instruction</td>
<td>0.3 s</td>
<td>130 %</td>
<td>1185 %</td>
<td>100 %</td>
</tr>
</tbody>
</table>

Table 5.2: Results synthesis
At a first glance in the table above, we can see that, although the values of the prediction error in percentage terms are higher than one would like to, these don't give the same interpretation of the systems' capacities than the decision accuracy results. While most of the times the measurement in percentage terms allows an easier interpretation of the metrics, sometimes it is important to use the values in units in order to have a correct understanding of the situation and the determine relevancy of those values.

When talking about the average prediction error in seconds for both scenarios, we have 8 and 0,3 seconds respectively. With a more detailed view over these numbers we can see that these not only make sense, but are actually pretty acceptable when it comes to the prediction accuracy. In both scenarios the percentage error is influenced by the time scale on which some measurements were made. For example, if we analyse the results from the second scenario, we can see that in spite of having no error measurement above one second, the SMAPE is really high. This happens because the formula removes any scale from the computation so that it may return an interchangeable result in percentage terms. This value though, does not have a real meaning to our situation and is therefore ignored.

If we look at the tables' first and last columns, which respectively correspond to the MAD and decision accuracy values, we can be confident that our system's prediction capabilities are good enough for it to make correct offloading decisions in either scenario. In fact, as expected, the prediction error of the first scenario is bigger than the second one, which is resultant not only from the fact that various predictions are made from values previously predicted, but also from the existence of "for" loops in the R-Benchmark 25's code. This increase though, did not show any significant influence on the decision accuracy.

What did cause the 25% reduction of the decision accuracy was the overestimation of the environment size after the execution. This is particularly prejudicial to those volunteers whose main characteristic is the communication speed they provide (network bandwidth).

Moving on to the prediction overhead, we see that for the multi instructions scenario, our system, on average, adds 16,2% to the local execution time. This increase while higher than expected does not compare to the glaring value of the single instruction scenario, where the average increase was of 1185%. In order to understand why these values were so high we need to inspect the scenarios.

Going back to figure 5.6, we can see that the prediction overhead for the Programmation (III) execution is significantly bigger than the ones obtained for the previous three measurements. This happens because the code of this benchmark is mostly composed either by assignment instructions or by functions calls whose execution is performed within half a second. Since our prediction is based on a function granularity approach, its overhead is expected to be correlated with the number of functions that the given code contains.

Figure 5.10 shows a comparison of the total overhead (in seconds) against the number of functions for each of the eight usage cases that we presented in this chapter. Here we can see a clear correlation between the two lines.

If the number of functions increases but the correspondent execution time does not increase nearly as much, then the overhead of our prediction will increasingly burden the execution since it becomes a
big percentage of the time necessary to finish the task.

The huge prediction overhead percentage values we see in the chart of figure 5.9 are along the same lines of what we just noted, i.e, the percentages are high not because the overhead was that much bigger but for the simple fact that the real execution time is almost zero. When looking to figure 5.10 we can see that in fact the last four points show the same behavior, in spite of the `crossprod(a)` example having a much smaller prediction overhead percentage. This, means that our system presents a minimum overhead of 0.8 seconds, which is cleary not suitable for basic instructions which take less than half a second to operate as is the case of the last three cases.

As mentioned throughout the document, this system uses an heuristic approach, and like every approach that relies on heuristic metrics, it is possible to find cases where the system works really well and others where the results are less optimistic. For the purpose of evaluating the system, we’ve used the R-Benchmark 25 script as representative of R’s usage, whose execution does not rely heavily on flow control elements.
Chapter 6

Conclusions

The main goal of this work was to present a prototype of a system capable of deciding whether an R program should be offloaded to a remote computer, and if so, select from a pool of remote volunteers which one provides the most benefit. The solution features a combination of execution time estimates with information relative to the remote volunteers, which is obtained through external communication. Furthermore, the solution was designed so that the decision process adds the least possible overhead to the execution, without overlooking the prediction accuracy.

The benefits of such offloading system include the extension of both the local memory and processing resources, but in this work we addressed only the processing aspect. We focused on a time based decision, where we choose the execution site by comparing the local execution time estimate against the ones from the volunteer nodes, and selecting the option that takes the least time.

The execution time is predicted by using a history based approach, where we use previous observations of function calls to reach an estimation value. Because not all functions are equal in behavior and syntax, we understand that there are functions over the thousands of packages that R provides that may not be correctly estimated by system, but based in the results presented, we are encouraged and believe that our approach to the prediction process is able to deliver good estimates for the most part of those functions. Particularly, when taking into consideration that R’s main focus is data analysis, and, because of that, the most complex functions are expected to be the ones that handle large amount of data, we are confident that our offloading decision system is able to perform as expected, accuracy wise.

Although we were capable of performing accurate decisions, the prediction overhead left some room for improvement. Having adopted an algorithm to estimate the execution time with a function granularity, we were unable to scale our system’s overhead by the time a given R script takes to execute, but instead by the number of functions that that script contains. This means that, for those situations where most of the expressions take almost no time to execute, all these functions essentially act as overhead, since even though they add no significant value to the total estimate value, they also need to be predicted, increasing the total overhead value.

Overall, in spite of needing some extra testing to find the real extent of its capabilities, this system and its approach to the offloading decision showed encouraging signs that, for regular data analysis
scripts (in R), it is a valid option.

6.1 Future Work

Since this system is not a finished product, there are several possible improvements that can be done. We have identified a few major topics.

Reduce Prediction Overhead

Our major concern here is to reduce the overhead, so that our system's impact in the total execution time is the least possible. Here we have two suggestions that may help that purpose, considering that the datasets have enough entries to be representative of the function's usage.

- Have models already computed for the most common function call sintaxes, i.e, identify the most common combination of arguments and have them modeled and prepared to be used. This way it would not be necessary to iterate through all entries to discover which are the ones that should be used for the prediction. Of course any function call that has a combination of arguments that do not match the ones previously computes would require it to be done anyway.

- Create a table with the function names of the functions which never took over one second to execute. This way one could identify, in a first stage, if the R script has any function function call that may possibly require offloading.

Improve Prediction Accuracy

While evaluating our system we identified that we are overestimating the final environment size. In order to better perform this estimate we can account for the variables that were already existent in the environment and are assigned to a new value. By removing the correspondent size of the previous version from the total estimate, it should be possible to get a better estimate of the real size.

Increase Complexity of Decision

While in this system we only consider the execution speed as a deciding factor to determine which machine brings greater benefit, in a real life situation, the user may give more importance to other attributes like the Mean Time Between Failures (MTBF), the machine reliability, or even the price (if he just wants the cheapest machine, ignoring the rest of the characteristics)

Free Up RAM

Free up RAM once the remote execution is initiated. This could be achieved by removing every variable from the local process's environment and loading them back from the updated environment once the remote execution finished.
Bibliography


[45]: The Center for Spatial Data Science. http://spatial.uchicago.edu/


[50]: CRAN - Contributed Packages. https://cran.r-project.org/web/packages/

[51]: Debian Packages of R Software. https://cran.r-project.org/bin/linux/debian/index.html


[55]: rPython R package. http://rpython.r-forge.r-project.org/

[56]: iPerf - Test tool for TCP, UDP and SCTP. https://iperf.fr
