Computational Cost Estimation using Volunteer Computing in R

Ricardo Simon Moreira Wagenmaker
ricardo.wagenmaker@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2017

Abstract

As organizations realize how data analysis helps them to harness their data and use it to identify new opportunities, specialized programming languages like R got their popularity risen. R is an open source programming language and an environment for statistical computing and graphics, whose increasing notoriety has attracted lots of new users. This increase includes 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 as an utility to measure the R's performance under different machines.

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

1. Introduction

As organizations realize how data analysis helps them to harness their data and use it to identify new opportunities, specialized programming languages like R got their popularity risen.

R is an open source programming language and an environment for statistical computing and graphics, whose increasing notoriety has attracted lots of new users. This increase includes 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 such VC platform 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 as a mean to select the best option out of the available remote volunteers.

In order to address these requirements and contribute for the development of the mentioned VC platform, this work proposes a decision system 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.

2. Related Work

The decision in our system comes up to four major time periods: the time it takes to predict performance, program's execution time estimate, 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 and what are its main characteristics.

2.1 Program Performance Analysis

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 [12], cluster and cloud [8] and networking [18, 22].

Ranging from the discovering of execution-time bounds in [23], to simpler pure history based analysis [6], passing by scalability predictions *empirical computational complexity* and program similarity measurements, there are several approaches that have achieved good results in analyzing the performace of computer programs. These 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.

A more recent approach, that focuses their efforts not only in the accuracy of the framework, but also in getting low overhead when generating the predictions, is Mantis [10, 17]. Mantis is a framework for predicting program performance on given inputs, and combines techniques from program analysis and machine learning.

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 [21]. 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, helping improve overall performance.

When comparing the two contexts mentioned earlier in which this technique may be used, their focuses have one fundamental difference: battery saving. Whereas in a mobile context, one of the main goals is reducing energy consumption and extending the battery duration [20, 25], in grid computing, the goal is solely improving performance, i.e, reducing the time required for execution [24].

2.3 Network Bandwidth Estimation

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.

Proposed by Keshav in [15], there are several approaches that use active probing [9, 11]. But more lightweight systems have also been developed using passive measurements [14, 19]

All techniques though, use some implementation of a filter to account for the noisy observations, and get a network bandwidth estimate value. One common aproach is to use the *exponentially-weighted moving average* (EWMA) filter.

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.

Although these systems work well in their context, neither property is advantageous all the times. [16] 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.

2.4 R programming

The R system is divided into is divided into two conceptual parts, having it's functionality divided into modular packages.

1. The "base" R system from the oficial repository CRAN [7]

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, grDecices, grid, methods, tools parallel, compiler, splines, tclk 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 reposiories, where users share their contributions in a worlwide community. The biggest repository is CRAN, which is the official repository and currently has over than 11000 available packages [1]. 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 [2].

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. It 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.

3.1 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.

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 (2.4) to generate a final estimate value. This means that we use a prediction technique with a function granularity.

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.

In this work we argue that a given function call has some arguments that are essential to the execution's behavior and some that are complementary. Therefore we estimate the complexity of a function call using a selected group of arguments, which we consider to be the arguments that are supplied without a corresponding tag.

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. Throught 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 syntatic 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.

3.2 Architecture

The designed system is composed of three major components: the RExecutor, the Market Infomation and the Incubator, each one operating at different execution stages with the objective of achieving the fastest decision process possible.

Figure 1 shows the system architecture.

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 . This operation can be seen in Figure 2, which shows the workflow of the live prediction process.

It starts after a user has given the command and finishes after the offloading decision has been made. As seen in figure 3.1, this module is composed by two inner modules: Complexity Estimator and Decider.

The Complexity Estimator performs a recursive analysis, which we call "Walking the Call Tree", where the main idea is to recursively advance in the code's call tree until we find the next function invocation.

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

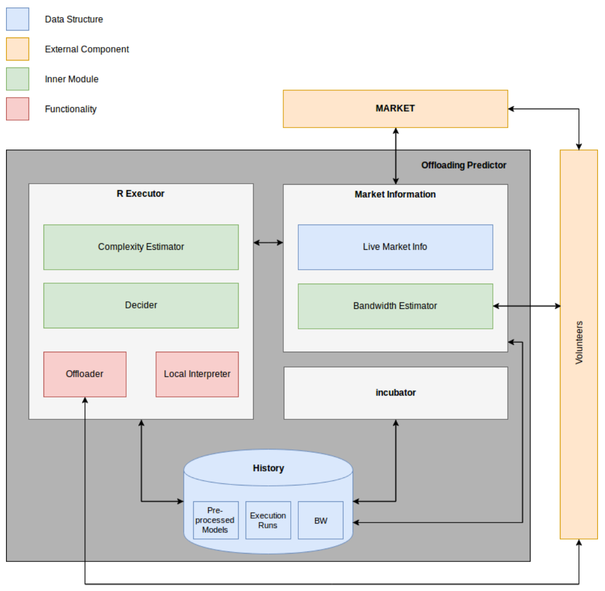

Figure 1: System Architecture

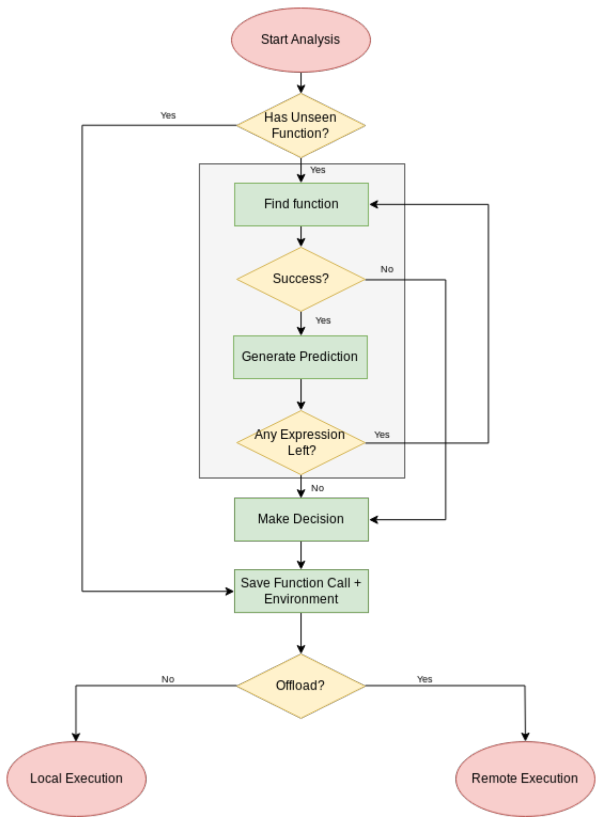

Figure 2: Live Workflow

– Use the control structures and the assignment operators (`<-` and `<<-`) to give context to the analysis and correlate expressions

– Use closure and builtin functions 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.

Every time a new function invocation is found, its execution time and size of its return object are predicted, and added to their respective total value.

In order to generate such estimates, the system begins by creating a new arguments profile of the fresh function call being predicted. Then proceeds to perform a history lookup, where are identifed all the dataset entries correspondent to any function calls that were supplied with the exact same arguments as the new function call. Furthermore, while going through every entry, it identifies which are the most important 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 where the object type is a collection (e.g. lists, vectors, matrix, etc), while the high significance variables are the ones where 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.

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.

Every time a prediction is requested, two different estimate values are computed, representative of the invocation's behavior: execution time and return object size.

Next, we create two linear models of two dimensions, which are always composed by two sets of values: 1. dependent variables(y) - ExecutionTime or ReturnObjectSize; independent or predictor variables(x) - MemorySize or NumericValue. Depending on the arguments' type, we use either the computed 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. This metric indicates how good the data fits to the equation.

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.

The Decider is then encharged with performing the offloading decision based on the information provided by three components:

– **Market Information.** Provides the user's current balance, the *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 we just made with a predefined offloading threshold (in seconds). If it's 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 what machine the program should be executed, the system begins by using equation 2 to estimate how much time the main expression is going to take when ran on a given remote machine (P_{remote}). This equation applies the performance score (S) to the local execution time estimate (P_{local}) we just calculated in the previous step (view workflow of figure 2).

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).

Then out of those selected machines, the decider picks the one that provide the lowest estimate of the total remote execution time (equation 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_{send}); 2- time to execute at the remote site; 3- time to retrieve the results ($T_{retrieve}$).

$$P_{local} = \sum Prediction(FCall_{non-special}) \quad (1)$$

$$P_{remote} = P_{local} * S \quad (2)$$

$$Cost = P_{remote} * n^{o}credits/timeunit \quad (3)$$

$$Ptotal_{remote} = T_{send} + P_{remote} + T_{retrieve} \quad (4)$$

If no affordable machine is found, then the code is executed locally. Otherwise, the necessary information is provided to the Offloader and the remote execution is initiated.

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 1. This external component is assumed to be reliable and secure, so all information provided is accurate.

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, Score, Mean Time Between Failures (MTBF).
- **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 does so in a periodical fashion, updating the values every previously defined time interval, 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. 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 (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.

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.

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. to generate new argument profiles for every function call. Afterwards, we execute the calls themselves with the arguments given within the correct enviroment. 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. 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.

4. Implementation

The system 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 in divididng 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 such integration, we've decided to implement the system's interface like a regular function, allowing the user to take advantage of our system like he would call any function. After the invocation, where the R expression is supplied, the decision operation is done through scripts that automate the process.

4.1 Component Communication

While most part of the communication between the components in our architecture (figure 1) is done through simple invocations(all within their own process), there are three that are more complex:

1. RExecutor `<-->` Market Information ;
2. Market Information `<-->` Market ;

3. Market Information <--> Volunteers ;

The Market Information is the component that acts as an intermediary between the local system and the external components, acting in a three-way front. It operates in a in a Python process with two threads of execution, 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 every predefined seconds.

The inner part of the communication (first item) refers to a communication 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 [5], 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 convertion 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 listenning 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 [3]- 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, so that our system by running an iperf client with the remote host's IP address and port, is able to get a diagnostic object of the network bandwith between the two machines.

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

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.

For the first scenario we used the R-Benchmark 25 script [4], which is globally used in the R community as an utility to measure the R's performance under different machines. We further decomposed it into four separate examples:

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

For the second scenario were 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 evaluation was performed with 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 was 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.

	Available Bandwidth	Performance Score	Credits/Unit	Balance in credits
User	-	-	-	●
Volunteer 1	1Mb/s	0,5	●	-
Volunteer 2	5Mb/s	0,75	●	-
Volunteer 3	10Mb/s	2	●	-

Table 5.1: External Components's Information

In order to evaluate our system we used the Prediction Overhead Percentage, the Simetric 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. The following two metrics are two of the most renown forecast-error metrics [13]. 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.

$$SMAPE = \frac{2}{N}\sum_{k=1}^{N}\frac{|F_k - A_k|}{F_k + A_k} \tag{5}$$

$$MAD = \frac{1}{N}\sum_{k=1}^{N}|F_k - A_k| \tag{6}$$

5. Results

	MAD	SMAPE	Prediction Overhead Percentage	Decision Accuracy
Multi Instructions	8 s	13,4 %	16,2 %	75 %
Single Instruction	0,3 s	130 %	1185 %	100 %

Figure 3: Results Synthesis

When talking about the average prediction error in seconds for both scenarios, we have 8 and 0,3 seconds respectively. 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 so 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.

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.

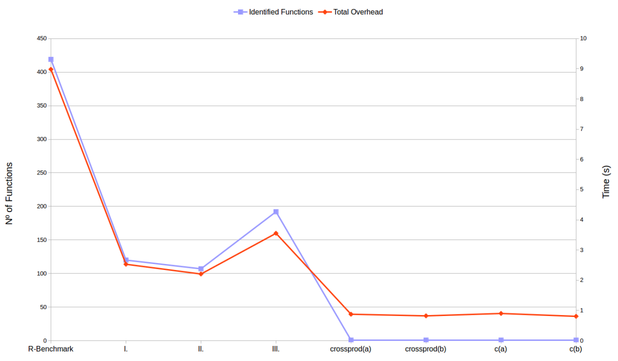

Figure 4: Prediction Overhead Behavior

Figure 4 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 average overhead percentage value we see in the synthesis table of figure 3, and whose measurements are graphically representated in the chart of figure 5 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 4 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

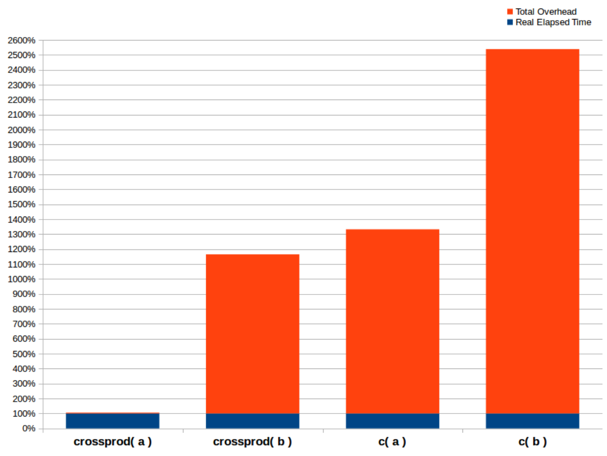

Figure 5: Prediction Overhead Percentage - single instruction scenario

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.

6. Conclusions

In this work we present a prototype of a system capable of deciding whether an R program should be offloaded, and if so, select from a pool of remote volunteers wich 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 overhead possible 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 sintax, we understand that there are functions over the thousands of packages that R provides that may not be correctly estimated by system, but based on 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.

Altough 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, inspite 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.

.

Acknowledgements

First and foremost I would like to sincerely thank my advisor Professor Joao 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 have not 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.

References

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

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

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

[4] R benchmarks. http://r.research.att.com/benchmarks/.

[5] R Language Definition. http://rpython.r-forge.r-project.org/.

[6] Some simple forecasting methods, The forecaster's toolbox. https://www.otexts.org/fpp/2/3.

[7] The Comprehensive R Archive Network. https://cran.rstudio.com/.

[8] P. Bodık, R. Griffith, C. Sutton, A. Fox, M. Jordan, and D. Patterson. Statistical machine learning makes automatic control practical for internet datacenters. In *Proceedings of the 2009 conference on Hot Topics in Cloud Computing*, pages 12–12, 2009.

[9] J.-C. Bolot. End-to-end packet delay and loss behavior in the internet. In *ACM SIGCOMM Computer Communication Review*, volume 23. ACM, 1993.

[10] B.-G. Chun, L. Huang, S. Lee, P. Maniatis, and M. Naik. Mantis: Predicting system performance through program analysis and modeling. *arXiv preprint arXiv:1010.0019*, 2010.

[11] A. B. Downey. Using pathchar to estimate internet link characteristics. In *ACM SIGCOMM Computer Communication Review*, volume 29. ACM, 1999.

[12] C. Gupta, A. Mehta, and U. Dayal. PQR: Predicting query execution times for autonomous workload management. In *Autonomic Computing, 2008. ICAC'08. International Conference on*, pages 13–22. IEEE, 2008.

[13] R. J. Hyndman et al. Another look at forecast-accuracy metrics for intermittent demand. *Foresight: The International Journal of Applied Forecasting*, 4(4):43–46, 2006.

[14] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, volume 18, pages 314–329. ACM, 1988.

[15] S. Keshav. *A control-theoretic approach to flow control*, volume 21. ACM, 1991.

[16] M. Kim and B. Noble. Mobile network estimation. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 298–309. ACM, 2001.

[17] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek. Mantis: automatic performance prediction for smartphone applications. In *Proceedings of the 2013 USENIX conference on Annual Technical Conference*, pages 297–308. USENIX Association, 2013.

[18] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. G. Greenberg, and Y.-M. Wang. Webprophet: Automating performance prediction for web services. In *NSDI*, volume 10, 2010.

[19] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 276–287. ACM, 1997.

[20] A. Pawar, V. Jagtap, and M. Bhamare. Time and energy saving through computation offloading with bandwidth consideration for mobile cloud computing. In *Proceedings of the Third International Symposium on Women in Computing and Informatics*. ACM, 2015.

[21] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal communications*, 8(4):10–17, 2001.

[22] M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, and M. Ammar. Answering what-if deployment and configuration questions with wise. In *ACM SIGCOMM Computer Communication Review*, volume 38. ACM, 2008.

[23] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, et al. The worst-case execution-time problem : Overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[24] R. Wolski, S. Gurun, C. Krintz, and D. Nurmi. Using bandwidth data to make computation offloading decisions. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008.

[25] F. Xia, F. Ding, J. Li, X. Kong, L. T. Yang, and J. Ma. Phone2Cloud: Exploiting computation offloading for energy saving on smartphones in mobile cloud computing. *Information Systems Frontiers*, 16(1):95–111, 2014.