UiO : Department of Informatics
University of Oslo

Pi And The Sky

Using Smart Offloading to Improve Performance on Low-cost Computers

Jun Ou
Networking System Administration
master thesis spring 2013

Pi And The Sky

Jun Ou
Networking System Administration

23rd May 2013

Abstract

This thesis explores a feasible way to implement an efficient interface for offloading computations from affordable computing device to external server in order to achieve increased application performance. Some tests have been done and the results show that the accomplished API is able to realize better perceived performance, and the optimization process has developed API into achieving smart offloading, a mechanism of adaptively determining offloading or not regarding practical situation. The findings in the thesis confirm the potential of utilizing computational offloading on affordable device as an economical method for providing a better user experience.

Acknowledgement

First and foremost, I would like to express my deepest gratitude to my supervisor Kyrre M. Begnum for his all support and help in the entire project period. His extraordinary talent in System Administration area provides me a great and valuable idea for this thesis. And then his patient instruction every week and continuous encouragement motivate me through all difficulties. Without his brilliant guidance and great abilities, this project would never have been the same. Thank you that you are always available when we need your help, thank the discussion meeting every week and you have to bear my poor English.

Secondly I would like to offer my sincere gratitude to our teacher Æleen Frisch, who recommend me to my supervisor in the first place. Thank you for your generous help in improving my paper writing, and we are so lucky to have you to be our teacher.

Also I would like to thank the Department of Informatics in University of Oslo for offering this Master program and providing me useful skills to become a System Administrator. Moreover, I owe my sincere appreciation to Oslo and Akerhus University College of Applied Science, those brilliant teachers and facilities offer us perfect studying environment.

Last but not least, I want to thank my most beloved family for supporting my study and always talking to me with great silent concern and care on the other side of phone. I am also grateful to my dear friends around me, especially to Kaihua Li and Sichao Song for their kind help on my thesis writing. In addition, I want to thank my boyfriend for supporting and encouraging me all the time, and I really appreciate that.

Thanks again, to all of you.

Oslo, May 2013
Jun Ou

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

It is now a cliché to state that computers have become an essential part of everyone's life. Computers continue to become faster, cheaper, smaller and part of more and more devices. People under 25 cannot even imagine a world without them. Statements like these are common and so often repeated that no one really thinks about them. But they are true only for people in the industrialized parts of the world: those regions and countries where industry is common and there is a significant infrastructure to support all of those computers.

There are many reasons why computers are not common in many places of the world:

- They may be too expensive.
- People may not have the knowledge required to use them.
- Their region may not have the stable power grid needed to run them.
- They may not solve the most pressing problems that people have.

There have been many efforts to make computers available and practical to more people in the world. Many people have looked for other novel ways that could reduce the cost in order to bring computing to users in less developed parts of the world. These kinds of efforts are seen as especially important in developing countries where it is essential for building a strong and independent economy that schools and colleges can teach computer science and increase local competence.

One of the most well-known efforts of this type was the One Laptop Per Child project[23] which attempted to provide rugged laptop computers to children around the world, at a cost of about $200 per laptop. A very recent affordable computing project is the Raspberry Pi. Many other affordable PC alternatives like this exist, and they are discussed in 2.1.1.

The Raspberry Pi is a credit-card size, single-board computer, equipped with miniature 700 MHz ARM and 512 MB of RAM in the newest model, which costs about $35. It also has very low power requirements and can even run for a time on AA batteries. It can use most ordinary television sets as a monitor. Even with such slow hardware by today's standards, the Raspberry Pi still enables users do many things that a normal PC does. It is capable of dealing with spreadsheets, word-processing software and simple games. The Raspberry Pi approach focuses on maximally simplified hardware to minimize size, power requirements and cost. It trades off computational power for these other factors, which can result in slow performance.

Although this performance level is low compared to today's high end devices, it may not matter in all cases. The Raspberry Pi may be adequate for many tasks for beginning users. Although it is not powerful when compared to other computers in use today, it is much more powerful than the early personal computers that were used at the beginning of the PC-era. For example, the Apple II computer, which sole for $2638 in 1977, had only a 16-bit processor running a 1 MHz with 143KB of memory. Nevertheless, it introduced computing to millions of children in the USA, in schools and at home.

When the native performance of the Raspberry Pi is not adequate for some computing task, it might be possible to augment it. If we could find ways to improve the performance of such affordable computing devices, users would get more out of the device and the rate of adoption might increase. Computational offloading is potential solution to the limited CPU capacity. This strategy consists of sending intensive computations to separate servers for execution so as to economize the use of the device's limited resources. If successful, this approach is capable of achieving significant efficiency of performance without affecting the low price of the computer very much since the external server could provide computation power for a large number of Raspberry Pi computers.

1.2 Problem Statement

Design and implement an efficient interface for offloading computations from the Raspberry Pi affordable computing device to an external server in order to achieve increased application performance.

The term 'computational offloading' indicates that the whole approach is a client-server distributed system. Specifically, the affordable machine is the client that is offloading the execution of data processing by sending requests to a server through the interface. The separate server focuses on generating workloads for processing requests and then delivering results back.

The interface is actually an application program interface (API) that enables application in the affordable machine device to communicate with the external server. The external servers can be a variety of different kinds of machines with strong computing capacity relative to that of the affordable computing device. These servers can be local or remote. It could also be organization self-manage or delegate-manage. Furthermore, when it comes to serving numerous poor machines, the amount of effective machines also demands increasing based on the number and frequency of requests.

Since the interface is an API, only compatible applications will be able to make use of it. It is not a solution which will transparently increase performance for all applications on the affordable computing device.

Once implemented, it will be necessary to compare performance with and without the offloading. This must include the perceived results of the user experience. One of the aims of the project is speeding up the perceived performance and providing a better user experience.

Finally, the interface design will require some care to achieve efficiency. It will incorporate optimization mechanisms rather than merely providing simple offloading. Since offloading consumes some network resources as an inherent cost, the optimization consideration attempts to avoid polluting the network with small requests which consume resources that are a poor tradeoff with the benefit they provide. We also expect that offloading will not always be the best choice for all computing procedures since the communication in client-server designed system required a certain period of elapsed time as well. Based on the research and analysis, we hope to identify the limits and crossover points when computational offloading is and is not desirable.

Chapter 2

Background

2.1 Affordable Computers

When the PC era has began in the early 1980s, the cost of a computer is always above $3,000. As the development of computer technology, a ordinary notebook nowadays has already been more than hundreds of times functional while only in the cost of under $400. But for many people around the world and large organizations with low budget, that price is still not acceptable. A new generation of low-cost mini computers can put an entire world of computing power in the palm of hand as little as $25[7].

2.1.1 Different Affordable Computers

Among all these low-cost computers, Raspberry Pi could rank first according to its extremely cheap price while cost effective. In the newest version Model B of Pi, the CPU is a 700 MHz ARM1176JZF-S core and the 512 MB SDRAM is shared with the GPU. It could achieve high-performance video and graphics on a single-board computer and thus makes itself a excellent media centre.

The VIA Technologies' APC could be the alternative of Raspberry Pi, also with single motherboard measures 17*8.5 cm and cost about $50. It runs a custom Android system, built for keyboard and mouse input, and includes a full set of consumer I/O ports that can be plugged directly into PC monitor or TV[2].

As the cost of computer components continues to drop, the Raspberry Pi is not the only inexpensive PC capable of running Linux any more. Inspired by the imitation wave, the Mele A1000, an ARM PC only for $70, has already out in the market and be assembled more components than Raspberry Pi-including a SATA port, a case and a faster processor[19].

There are some other affordable computers similar to Raspberry Pi as well, such as Aakash and Ubislate offered by Datawind Ltd.[6] for $40 and

$60 separately; MK802 Andriod Mini PC for $74 developed by company Miniand[20].

2.1.2 Projects with Raspberry Pi

In this paper, the Raspberry Pi is adopted in our research as typical affordable computer because of its popularity and the cheapest price. And in 2012, it is a big first year for this $35 mini Linux PC. As soon as they started shipping, these makers all around the world with great innovation passion were eager to get their hands on the pocket-sized computer to realize their DIY dreams. In just few months, various great Raspberry Pi projects have been working through and it is no doubt that more cool stuff is coming in 2013.

Raspberry Pi Web Sever

In so many projects, setting up a web server with Pi and making it work right is not very different from other Linux machines. After installing and configuring the custom Debian image for Pi, Raspbian[28], the firmware and software demands up to date. Then the popular web server program, Apache in this case[21], is deployed by the author. This is a fun experiment for web server installation, configuration and testing, but not a decent option for hosting any commercial Website from the author's recommendation.

Since Apache is probably not the best option as Webserver on Pi, the same Pi enthusiast developed speed tests for different Webservers to compare the performance of each server on low powered hardware. His experiment was designed as below[22]:

4 pages for tests	Small Text Test - html page 177 bytes (small, quick transactions) Large Text Test - html page 95,881 bytes (large, long transactions) Small Image Test - Small PNG load (849 bytes) Large Image Test - Large JPG load (179,000)
4 softwares	Apache Nginx Monkey HTTP Lighttpd

Table 2.1: Pi Project on Webserver

From analysis on results, the overall conclusion came up that Nginx was proposed to be the fastest and most reliable Webserver solution on Pi

since it is more mature and has more speediness and stabilization.

Raspberry Pi OwnCloud

Raspberry Pi with some software called OwnCloud[24], could be used for building personal data center like the service of Dropbox[8] and it has been implemented in project[27]. All advance-phrase preparation is a Raspberry Pi, an USB external hard disk, an enclosure and wireless network card. Then following procedures are interpreted as steps[27]:

- setting up the network and give Pi a fixed IP address
- install and configure php & Apache on Pi for downloading ownCloud and accessing data files
- download and setup own cloud, then place it in the encloser

Audiobook Player

Another Raspberry Pi based project, called Audiobook Player[5]. The initial motivation of that is helping people who has impaired vision, such as old people. It is available to people that just prefer to do other things and having a audio book concurrently, read aloud for them as well.

This project has following features[5]:

- **always on** When the Raspberry Pi is on power, it will boot up and start to execute a self-written python script with the audio book in pause.
- **one button usage** The button enables the audio book being paused and unpaused. If it is pressed longer than 4 seconds, the audio book will go back one track.
- **remembers position** It always remembers the position played last time.
- **only one audiobook** There will always be only one audio book in the Raspberry Pi.
- **easy audio book deployment** When a USB thumb drive with special label is to plug into Pi, the audio book will stop playing and mount this drive. Then Pi replaces old audio book with the new one in thumb drive and rebuilds the play list. As soon as unplugging the drive, the new audio book starts in pause mode.
- **multi format** Since it uses music player daemon, the player supports Ogg Vorbis, FLAC, OggFLAC, MP2, MP3, MP4/AAC, MOD, Musepack and wave.

2.2 ICT in Education

Information communication technology(ICT) is widely used as same phraseology as information technology(IT). However, it is a particular description of communication integration, which always consist of telecommunications, computers, enterprise software and so on. These components generate a whole ICT system that renders users availability, storability and operability of information.

Global economic and social trends over the past several decades have profound implications for educational reform and the use of technology in schools[13]. On the contrary, quality education makes great contribution to economic growth likewise. Microeconomic data from 42 countries found that an average rate of return for an additional year of schooling was a 9.7% increase in personal income[26]. A cross-country macroeconomic study found that there was an additional 0.44% growth in a country's per capita GDP for each additional average year of attained schooling, a return on investment of 7%[3]. In some other studies come to that returns go as high as 12%[30].

The introduction of Information and Communication Technology(ICT) into education system is a part of the educational revolution as ICT is designed to serve as vehicle for improving efficiency of the educational process[12]. Thus the awareness of significance of both ICT education and the impact of ICT on education needs to be enhanced.

Although increased phenomenons illustrate that schools are trying to attract students by competitive ICT education environment, the current situation still appears there is no related policy to function specially as connection between education and professional community.

Moreover, taking the firm and decisive action to ensure schools and educational facilities have the resources they need, not always get smooth realization from policymakers and ministry officials. These resources include funding, staff, infrastructure and the training required for them to take advantage of what ICT has to offer in the educational environment[10]. Therefore the maximum utilization of existing limited ICT resources to provide quality education becomes more important.

2.2.1 Raspberry Pi used in Educational ICT Environment

The original motivation behind the creation of Raspberry Pi is all for kid's education. As the development and application of information technology, the way child interacting with computers has been changed, and the rise of home PC and game consoles programming replace that in old command line environment learned by earlier generation. Together with inadequate ICT curriculum, colonisation lessons on using Word, Excel or writing Web

pages, they lead to year-and-year decline in number and skill level of the A level students who apply for reading Computer Science in each academic year[1].

Thus Eben Upton and his colleagues at the University of Cambridge's Computer Laboratory, came up the idea of bringing affordable but powerful enough device to encourage kids learning programming, whose initial interests are not on purely programming-oriented device, and then made it into reality.

In the early of this year, Google provided 15,000 Raspberry Pi Model Bs for school kids around the UK[17], which is a generous and brilliant way to inspire those children having aptitude on computing to explore their capacities properly.

Another case of using Raspberry Pi for educational ICT environment is described by Miss Philbin in UK, a google certificated teacher who works hard to bring computing to her key stage 3 students of secondary school began from this January. The initial reasons of using Pi are omitting school network and workstation configuration, broaden students' understanding of computer hardware and how they work. In her teaching and learning journal, various problems are proposed, most of them are because of poor hardware, and she still solved some with the help from Pi foundation[25]:

- **Monitors and Adapters** Since the practical situation in UK is most monitors in schools are VGA while Pi only renders HDMI interface, so the first suggestion is deploying HDML to VGA adapters. However, using cheap HDMI to VGA adapters is at the risk of blowing Pi's diodes, hence Miss Philbin collected DVI monitors and then sourced HDMI to DVI adapters that functioning well afterward.
- **SD Cards, Images and Backing Up Work** Pi is unable to compatible completely with SD card usage that results in corrupting data. Moreover, checking produced work from students on SD cards by reimaging them repeatedly bring cumbersome process to teachers. This problem is not solved in her journal.
- **Cases** In Miss Philbin's class, she assembled Pimoroni PiBow cases to avoid Pi remain as naked board when it has been setting up and packed away.
- **Micro USB Power Supplies, USB Keyboards and Mice** These equipments could be collected easily and cheaply.
- **Storage** With the help from a team, the teacher could prepare all extensions cables and cover them by desk so that all students are capable of trying and use equipments in place, plugin and unplugin everything themselves.

2.3 Computation Offloading

When it comes to computation offloading in reality, there are lots of novel and in-depth researches have been investigated in last few years. Most of them are analysed and tested on another device with resource-poor hardware, mobile smart phones and called mobile computing.

Smart phones, a hand-held computing device, be able to realize the vision of "information at my fingertips at any time and place", that is only a dream in the middle of 1990s. But today, ubiquitous email and Web access is a reality and has been experienced by millions of users worldwide[29].

While from the user's view, a mobile device can never be too small, too light or have too long battery life[29]. And the longer battery life is the most desired feature among them. But it is obviously a crucial issue for power management since a singe user maybe prefer to run multiple application on mobile phones at the same time, that leads to limited battery be expended faster but the performance would never as good as on the device with static hardware.

Despite of several known power-conservation methods like turning off the hand-held device screen when it is not needed, optimizing I/O, a partition scheme is constructed in [18], which profiles computation at the level of procedure calls and the computing device is connected to a more powerful server via LAN for offloading computation. Then a program could be divided into server tasks and client tasks so as to minimize consumed energy.

While computation workload and communication requirement may change with different execution instances and one fixed program partition decision would result in working poorly according to [31], so different program partition decisions also need to be made at run time when we have sufficient information about workload and communication requirement. In the [31], a parametric program analysis to transform the program is presented to achieve optimal partition decision based on run-time parameter values.

When the server with powerful capability on computing is into consideration, a popular option nowadays for mobile device is utilizing cloud computing. The cloud heralds a new era of computing where application services are provided through the Internet[15]. It is available through many companies, such as Amazon, Google and VMware. The shared infrastructure of cloud computing works like a utility that the customer only pay for what they need. As such, the recent Cisco report predicted that worldwide cloud traffic will explode in the coming years, growing six times in size by 2016.

But from the analysis in [15], it suggested that cloud computing can po-

tentially save energy for mobile users, while not all applications are energy efficient when it is migrated to the cloud. Cloud also has its limits for mobile device especially when it needs to execute a resource-intensive application on a distance high-performance compute server since long WAN latencies are a fundamental obstacle[29]. However, the performance of cloud service varies significantly between mobile computing and desktop since it must save energy for desktop. Moreover, the other energy cost in service for privacy, security, reliable, and data communication are also considerable before offloading[15].

Since the cloud service not always enable energy saving on all applications because of the increased latency by distance, the alternative mechanism is offloading the computation to a nearby server or resource-rich cloudlet. But it is still hard to decide whether it is worth to adopt offloading as many typical single-purpose applications on smart phones could run easily within its own resource and offloading leads to increased running time because processing distributed program, like profiling, optimizing, migrating, is also a complex procedure.

So face to various tasks and programs, different decisions are needed to make out. In paper[9, 11], model for predicting the performance of distributed programs, that achieves real-time adaptive offloading, has been put forward. They uses different models to define problem, and then implement algorithm for update model or code offloading. The application quickly adjusts some parameters and minimizes the difference between predicted and measured performance adaptively. The accurate prediction helps determine whether to offload tasks and gain the expected performance improvement at the same time.

While in[32], another approach is proposed in which does not require estimating the computation time before execution. The program is executed on the portable client with a timeout first and if the computation is not completed after the timeout, it is offloaded to the server. This mechanism demands to collect online statistics of computing time and find out the optimal timeout. Then the result of further experiments shows that these methods can save up to 17% more energy than existing approaches.

2.4 API

An API(Application Programming Interface) is used as an interface to enable the communication between software components. The API in our specific assignment is implemented by XML-RPC protocol, a remote procedure call mechanism(RPC), enables data coding with XML and HTTP transport mechanism.

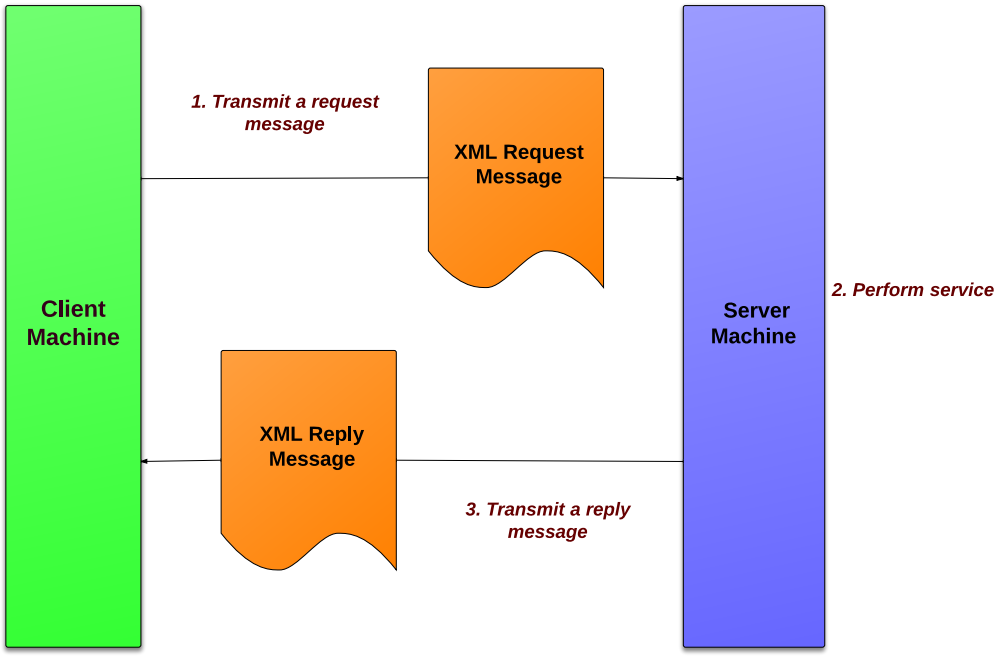


Figure 2.1: XML-PRC Model

The XML-RPC model is shown as the figure 2.1 above. It could be regarded as a distributed server-client system to deal with computing tasks and provides increased performance to users.

Chapter 3

Approach

As noted in the problem statement, this project aims to improving computing performance on affordable machines. And based on discussion in the previews chapter, excluding the method of strengthening hardware performance, as it is unable to maintain the price advantage, we would consider using the methodology of computational offloading. The conclusion that computing offloading could achieve increased performance under certain conditions is in our expected range, the key difficulty this research faces is to analyze limitation and boundary of utilizing the mechanism in realistic situation from different aspects, then explore the optimization approach to minimize performance cost.

The project could be split into three parts, and each of them would be interpreted in detail in following sections:

- **Setting up interface:** The API that supports computing offloading from Pi to server needs to be built up. There are various methods providing computational offloading feature for Pi, but this project demands the API usage be combined with actual fact that it would work for educational ICT environment.

- **Performance comparison and analysis:** After achieving feasible API, different experiments on API utility would be proposed and tested. Then some particular data are chosen to describe computing performances with and without using API. In all those picked metrics, runtime would be regarded as the most important one as it explains the perceived performance for users directly.

- **Optimization exploration:** The comparison and analysis from experiment data could help us to refine the API design, but with the limited given time and resource constraints, the unpredictability of improvement optimization is apparent.

3.1 API Implementation

The first step of this project is producing a feasible API prototype and then implement it to realize computing offloading. As mentioned previously, our main goal is increasing the computing performance on Pi, which has appeared in the educational domain within novel usage in a fashionable way, especially for programming teaching. So the mechanism adopted to build API requires to be consistent with this practical situation as well.

Therefore the decision on programming language for developing API is the first priority. As mentioned in paper [14], using traditional languages like C or C++ would consume a lot of time in understanding the syntax, semantics and program design, the idea of algorithm programming teaching, which is one of fundamental requirements in basic Computer Science(CS) education, would be overshadowed as well. But for Python, it is a natural programming language close to pseudo-code and easy to learn, to use as well as to test students' implementation for enhancing their understanding of CS.

In addition to, when we look back to the initial desire of Pi creation and usage, school python teaching stands out exactly as the original purpose. Moreover, as a popular and strong programming language, python provides its own XML-RPC protocol modules that work as the foundation of API for users to take advantage of, which enable connection and communication between Pi and server.

3.1.1 Python for API

For the purpose of deploying Python modules supporting XML-RPC protocol to deal with computing offloading, the SimpleXMLRPCServer module is used to write XML-RPC server that enables a standalone HTTP server to listen for incoming requests and responding accordingly. Moreover, the xmlrpclib module is applied on client-side for supporting XML-RPC. These two modules could be used easily for achieving successful communication without worrying about the underlying encoding or data transport. The basic server-client system example with Python modules are established all on one machine as below[4]:

XML-RPC Server and Client

```
#Server side
import SimpleXMLRPCServer
import math

def add(x,y):
  "Add two numbers"
  return x+y

#set up local host as XML-RPC server
s = SimpleXMLRPCServer.SimpleXMLRPCServer(("localhost",8080),
allow_none=True)
#register function 'add' with server
s.register_function(add)
#register an object to resolve method names
not registered with register_function()
s.register_instance(math)
#add XML-RPC introspection functions
s.register_introspection_functions()
s.serve_forever()

#Client side
import xmlrpclib

s=xmlrpclib.ServerProxy("http://localhost:8080")
#s.add(1,2) returns 1+2=3 and assign 3 to a
a=s.add(1,2)
```

Since those tasks are mostly about math-related problems when taking Python programming teaching into consideration, thus the realization of API simplifies those time-consuming computation under certain condition. The limitation of API is obvious from codes above that the disability of providing services to all applications as it is implemented on procedure call level, only be influential for these applications capable of calling functions or importing objects registered with server.

Moreover, as the simple XMLRPC server built by Python described above, it only supports single thread for processing requests by default. Although it may add the system.mullticall() function to server as well, which could ask for handling multiple tasks in one request package, then the server will still deal with those tasks one by one without parallel communication. Since our experiment focuses on improving Pi's performance within API usage and adopts one Pi to one server as the basic test environment, thus the single-threaded server does not have much effect on results of tests but it demands to be developed for functioning in parallel in the future when putting it into reality using.

3.2 Test Design and Data Analysis

This section attempts to design typical scenarios to measure and investigate the API utility behaviors for further optimization. When setting API up in the back within achieving basic features, some trials could start to test offloading different functions to separate server as well as on Pi alone without offloading to generate comparison data for later analysis.

3.2.1 Benchmark Tests

Before testing the API utility, the raw system performance of both Pi and server need to be generated and described at first so as to get overall understanding in the variation caused by different hardware. Furthermore, the bottleneck finding for resource usage on them may have positive effect on later experiment design.The basic benchmark tests by benchmark tool 'Sysbench' are proposed as below:

Sysbench	Test Variable
CPU	–cpu-max-prime= 50,100,300,500,700,1000
Memory	–memory-block-size= 1K,50K,256K,1M,50M,128M

Table 3.1: Purposed Benchmark Tests

All these parameters selection, benchmark tool as well as test repetition time and recorded metrics for both Pi and servers, demand to be identical, so that differences in result are only caused by hardware variation.

3.2.2 Variables in API Utility

Since there are various variables influencing the offloading performance, preliminary investigation of the whole communication process is accomplished and interpreted by different terms in figure 3.1:

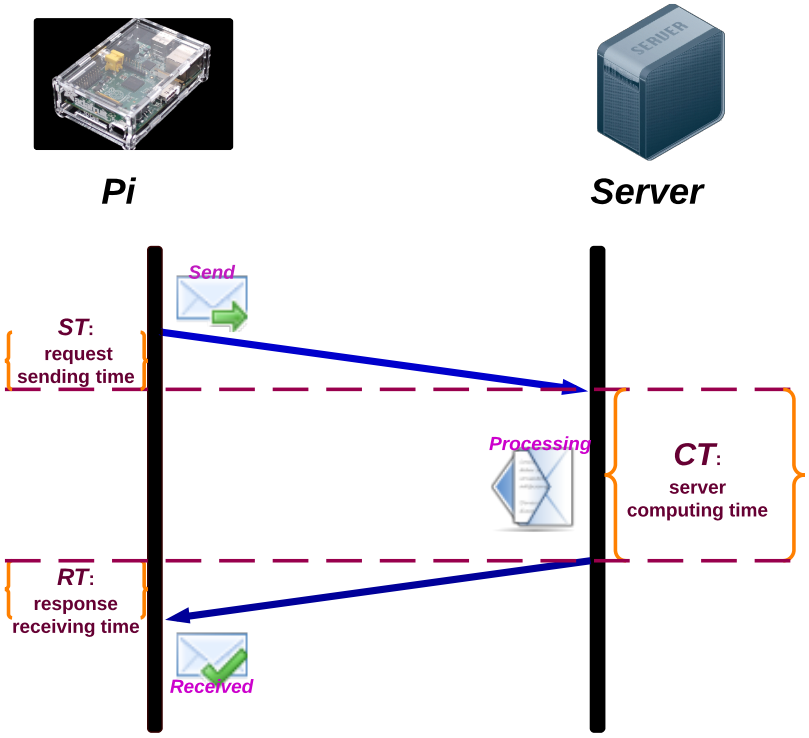


Figure 3.1: API Utility Prototype

- **ST** time spent on outgoing request from Pi to Server
- **RT** time spent on incoming response from Server to Pi
- **CT** time spent on processing request
- **RTT** round trip time for message transmission
- **RC** request complexity means how hard the computing would be
- **RS** request size, indicates the number of bytes in each request

All of these terms listed above could be used as considerable measurements for describing API performance, and there are evident mutual effects between them as well. The following content would analyze these connections in detail.

To start with, variables *ST* and *RT* would be distinct as the distance between Pi and server varies. Having been noted in 1.2, server as target to offload computing covers different kinds depending on practical condition, thus the strong capability rather than restraining existed form of machine

is under our consideration so as to improve the flexibility and utility of API.

Moreover, the networking status affects ST and RT as well, especially when the server is built on cloud as virtual machine, the networking condition would be more difficult to predict, hence the instability may result in networking latency differing from each same test in data collection process.

In addition to that, variable RS is influential to both ST and RT as more transmission time is consumed when workload gets heavier. As we are not going to explore the composition of XML-RPC request in detail, the RS would be explained as data size transmitted by each request in this project.

For variable RTT that stands for round trip time, it is the sum of ST and RT obviously as:

$$RTT = ST + RT$$

From the discussion above, RS and networking condition would impact on ST and RT, thus RTT could be interpreted as a function of RS and network proximity as:

$$RTT = f(RS, Network\_Proximity)$$

Then both server's computational capability and request computing complexity expressed by RC determine the variable CT, as hardware resource is a significant metric to investigate in this project. Our main purpose is deploying Pi with API for school teaching, and it is hard to conclude overall performance from different kinds of servers, so both strong server and normal desktop are our consideration. Thus same as RTT, CT could be described as a function in the following:

$$CT = f(RC, Server\_Power)$$

3.2.3 Experiment Scenario Design

Since so many variables would affect API utility performance and offloading for Raspberry Pi has never been tested before, the complexity of metrics as well as experiment type selection is apparent in this project. The testing rule we would follow is from easy test to hard one, from general test to specific one.

Metrics Selection

The main goal of our experiment is generating appropriate data with potential capability to display performance variation of Pi and server from API usage. In addition to selecting run-time as the most significant metric for API evaluation, it would be easy to collect performance data on the server part, as there are lots of monitoring tools available to deploy. However, it is unworthy of running monitor tool to trace CPU and memory performance of Pi, as it would occupy a certain amount of resource. And with extremely limited resource as well, Pi's computing performance on dealing with running experiment would be influenced seriously.

In order to monitor the experiment process without causing too many effects on Pi concurrently, the process performance data can be extracted from virtual filesystem '/proc' locates in memory dynamically.

Under the path '/proc/', each running process has its own directory named by process ID(pid). Then under each '/proc/$pid/', there exists various files storing all process information[16]. Since the CPU and Memory performance are what we are concerned about, the data in files named '/proc/$pid/stat/', '/proc/$pid/status' and '/proc/$pid/sched' would be fetched. Several important metrics in these data files are selected for further analysis and they are shown as table 3.2 below.

File	Metrics	
/proc/$pid/status	VmSize VmRSS	virtual memory size resident set size
/proc/$pid/stat	min_flt maj_flt	number of minor page fault number of major page fault
/proc/$pid/sched	sun_exec_runtime wait_sum iowait_sum nr_involuntary_switches	total runtime total wait time I/O blocking time number of involuntary switch

Table 3.2: Fetched metrics in '/proc/$pid/'

Files both '/proc/$pid/status' and '/proc/$pid/stat' show detailed memory usage information of process. In Linux system virtual memory consists of physical memory and swap locating on hard disk is used to provide running processes memory resource, and all pages stored in memory are mapped to virtual address space. So for a process, when it demands to access a page in memory, there are two possibilities for getting this page. If it is in physical memory, it could be processed by CPU very quickly; While when it is only mapped to virtual space but has not been loaded into real memory, a page fault occurs. A major page fault means this page has to be loaded in memory from disk and multiple major page faults would result in serious disk latency problem for the process. As we want to find out how intensive the memory has been used for tests and

whether the process running delay is caused by increased page fault numbers, both memory usage information and page fault numbers are recorded into our data files.

File '/proc/$pid/sched' stores the CPU scheduler information for process. Since the CPU resource of Pi is also limited, increased computing complexity brings in intensive CPU usage. Applying API into Pi is able to release some CPU resource for other process and extend the life of hardware.

Test Design

Since this project demands to generate comparison data for analyzing API performance in detail, each test includes two parts, baseline testing and API utility testing. In baseline testing, there is no API usage and all results are collected from Pi running alone. The second part of API utility testing would achieve increased performance to some extent predictably. The variation of these two parts is capable of explaining robustness as well as boundedness in API utility.

The general work flow of previous tests could be described as the figure 3.2 below. Given different input parameters, the first driver script experiment.pl would decide which kind of tests to execute, baseline or python code test, then outputs multiple data files in specified path. Another script analysis.pl is used for processing these data files to generate different tables and figures for better interpretation.

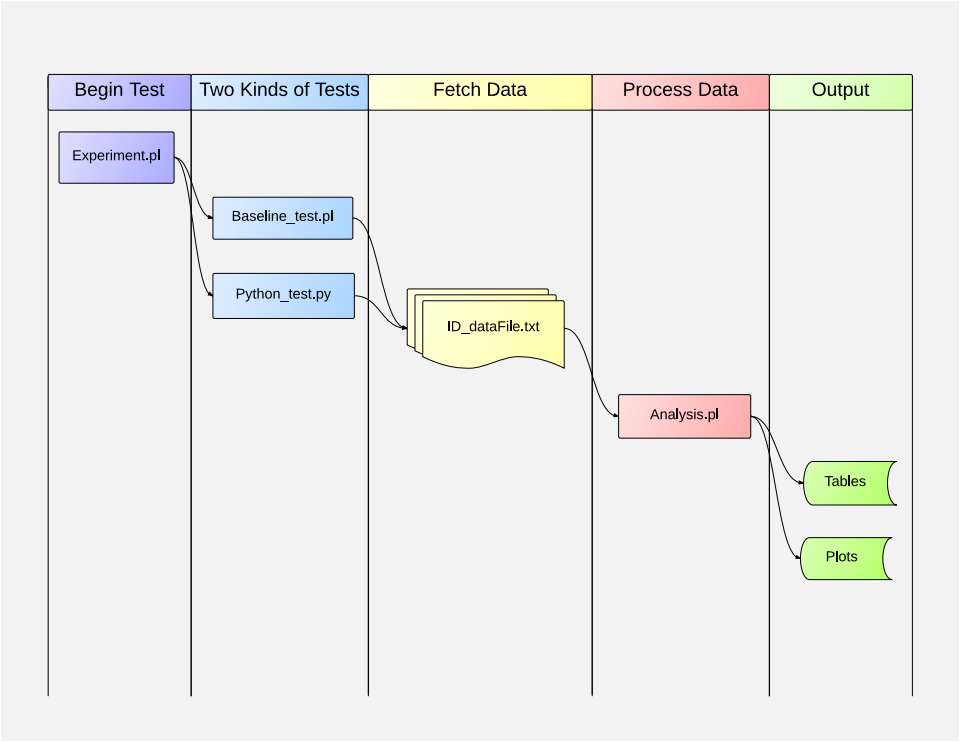


Figure 3.2: WorkFlow of Testing Baseline and All-offloading

When executing all tests in experiment, each kind of test requires running for multiple times to decrease systematic and random error, and the amount is usually 20 or more based on principle of statistics. Thus the result data could be more realistic and reliable for later comparison and analysis.

Scenario Expansion

As was discussed above, there are various servers that could be applied in this project according to realistic usage. So 12 different scenarios are proposed as table 3.3 shows below. In the table, there are four options for server choosing, Sl, Sr, Sld and Srd. It means local server, remote server, local desktop and remote desktop separately.

ID	Different server tests			
P2S	P2Sl	P2Sr	P2Sld	P2Srd
P*2S	P*2Sl	P*2Sr	P*2Sld	P*2Srd
P*2S*	P*2S*l	P*2S*r	P*2S*ld	P*2S*rd

Table 3.3: Purposed test scenarios

Despite of various servers selection, three different scenarios are shown

above towards real-life situations as school-scale increased, more Pis and servers might be adopted, thus simulation of these circumstances enables prediction of performance variation in order to get improvement.

- **One Pi to One Server(P2S)**

This simplest scenario will not happen in practical utility but is still typical for basic performance testing. Since compared with Pi, server would be much more powerful within holding enough hardware resource relatively, a certain number of Pi are not able to cause resource-constrained situation, hence this scenario, one Pi offloading computing to one server, would easily gain basic performance improvement in utilizing API.

Same as 3.2.3 noted above, uniform computing load would be generated both in baseline and API utility tests. Moreover, local and remote servers would also be tested separately. Then test execution time is collected as the main metric to show performance discrepancy.

- **Many Pis to One Server(P*2S)**

Since our API is developed for basic computer science teaching at school, this scenario is common that a number of Pis are used by different students, and then they offload all computing to a local or distant server. The mechanism is convenient and economic because it is possible to achieve overall increased performance on Pis within only paying extra money for one server maintenance.

These same tests in last scenario could be executed in this situation as well. As request number and computing complexity raise, handling computing process concurrently results in more intensive resource usage on server, so this scenario is able to reveal server's performance capability limit.

- **Many Pis to Many Servers(P*2S*)**

When the Pi adoption increased among education area, and a growing number of schools are willing to manage Pis and servers in a cooperative way, the third scenario is purposed to realize our API using in big education organization with deep connection.

A certain number of servers are managed to run in the back end, processing those offloading requests from different schools. The mechanism that sets up a load balancer in the front of servers could achieve request equal distribution for the purpose of protecting server regular operation from overload crash.

3.3 Optimization

When all tests have been done, some limitations may appear from performance data comparison. Since we want to integrate this API with real-life school basic computer and science teaching, further optimization design could bring better performance to students and increase practicability.

In paper [9], offloading applications for smartphones has quality of service requirements, so an adaptable offloading mechanism is proposed that works by constantly monitoring the time required to execute core service. Then the algorithm predicts whether offloading be advantageous on performance and determines services execution on smartphone itself or on surrogate device.

The adaptive approach is also mentioned in paper [11, 18, 32] while using different models for prediction and making offloading decision. This concept inspires us to improve the API fixed mechanism that offloading all computing needs from Pi to server, which might result in working even more poorly as the requirement of extra cost on transmission.

So it is predicted that some easy computing process executed on Pi locally rather than offloading may cost less time. The more flexible and dynamic solution of API utility for more performance will be part of the optimal API design.

Chapter 4

Results and Analysis

This chapter will describe all results of tests and analyze collected API performance data for later optimization. Then the whole optimization process is presented in the last section as well.

4.1 Machine Setting

As given time and resource for this project are limited, local and remote desktop are tested in our actual experiments. Moreover, the virtual machine on cloud was deployed as remote server since the cloud service is widely adopted nowadays.

Since the performances both on local and remote server demand comparison, the computing environment of servers requires identical setting up. However, the cloud restricts the changing allowance to virtual machines and their properties. Therefore, the physical machine has to compromise on the limitation of virtual platform and utilize similar hardware setting.

In reality, some machine setting properties are displayed in table 4.1, which shows that the system CPU and Memory resource of local and remote desktop are similar and powerful comparatively while the Pi is extremely resource-poor.

Properties	Raspberry Pi	Local desktop	Remote desktop on cloud
Processor	ARMv6-compatible processor rev 7 (v6l),700 MHz	Intel(R) Core(TM)2 Duo CPU E6550 @ 2.33GHz	Intel(R) Xeon(R) CPU E5507 @ 2.27GHz
MemTotal(kB)	189100	8037496	7629484
SwapTotal(kB)	102396	8253436	0
File System	/dev/root 7.3G /dev/mmcblk0p1 56M	/dev/sda1 139G /dev/sdb1 2.8T	/dev/xvda1 7.9G /dev/xvdb 414G

Table 4.1: Machine Setting in Experiment

4.2 Data Description

This section will present the five tests executed on Pi and server, a benchmark test and other four API utility tests, which show performance discrepancy in with and without applying API.

4.2.1 Benchmark Tests

The purpose of implementing benchmark tests for Pi and two kinds of servers is to get overall performance understanding. The tests focus on evaluating CPU and memory capacity performance, and they have been done by the benchmark tool 'Sysbench' within simple usage.

As proposed in table 3.1, CPU performance is measured by the time cost for calculating prime numbers up to a value specified by test argument. The same option is selected in each CPU test for the three devices: Pi, local and remote desktop. The CPU performance variation is displayed in figures 4.1 and 4.2, where the error bar is presented as 2 times of standard deviation:

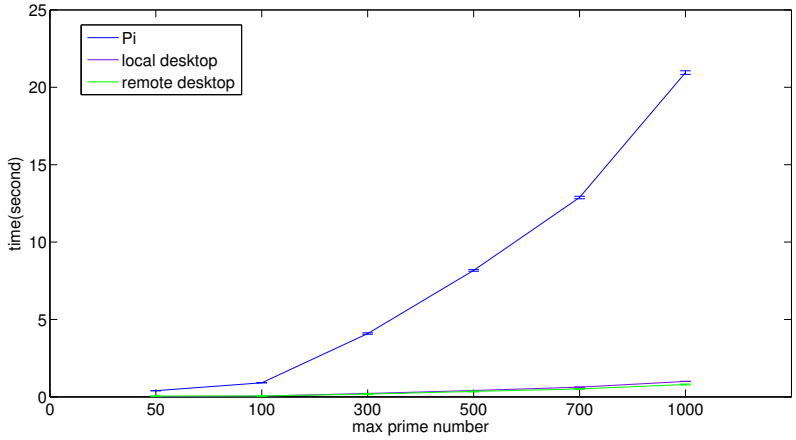


Figure 4.1: CPU performance in benchmark tests

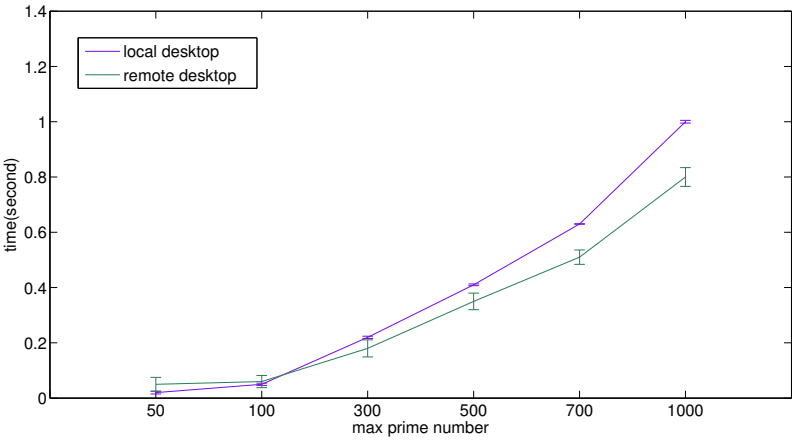


Figure 4.2: Servers' CPU performance in benchmark tests

In these two figures above, the first one illustrates how much more powerful the servers' CPU is compared to that of Pi. According to the retrieved data, it indicates that as upper prime value limit grows, the calculation time increases as well. However, Pi has faster growing of over 20 seconds, 20 times longer than the time spent on servers that is around 1 second regarding the largest prime value limit, 1000.

Since the CPU performances of both servers appear to be almost overlap in figure 4.1, figure 4.2 presents the discrepancy more clearly. It also shows that the remote desktop has slightly better capability when the prime value limit is greater than 100 while the performance is not as stable as that of the local one since the standard deviation appears larger obviously.

Same as in table 3.1, memory performance is evaluated by the transferring speed of writing a certain size of data with specified memory block.

We set a fixed total transferring size of data as 2 GB and select 6 sets value of block size for testing. The result is presented in figure 4.3 where two times of standard deviation is displayed:

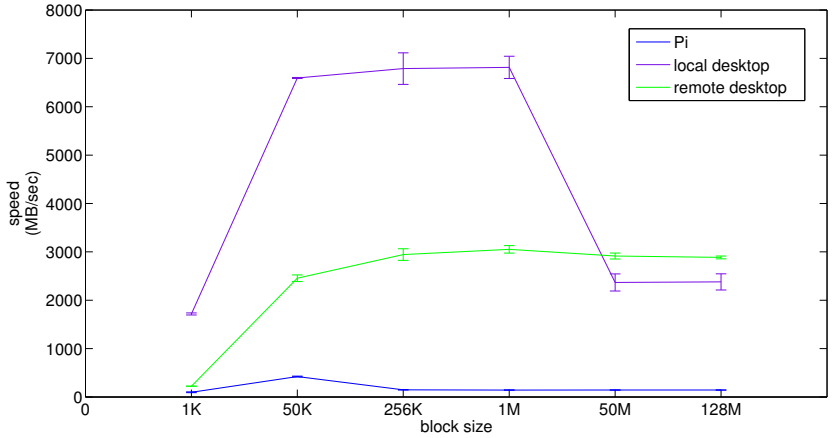


Figure 4.3: Memory performance in benchmark tests

The figure above illustrates the Pi's memory performance weakness compared to desktop. The fetched data indicates that the fastest memory writing speed of Pi is about 400 MB/sec when the block size is around 50 KB. While as the block size getting larger than 256 KB, the writing speed keeps about 140 MB/sec in a relatively stable way.

For servers, the local desktop has better capability than the remote one when block size is smaller than 50 MB. The fastest writing speed occurs on local desktop as block size increases from 50 KB to 1 MB, which is about 6700 MB/sec while the remote one is only about 3000 MB/sec at most. However, when the block size is larger than 1 MB, the writing speed on both of them decrease and local desktop reduces more fiercely, then it becomes even slower than remote against block size 50 MB. Besides, remote desktop has overall stable performance as its standard deviation is smaller than that of the local one.

In general, all benchmark results shown above illustrate that these deployed desktop are more powerful than client Pi, which implies stronger computing capability based on more valuable hardware. So it is predictable that Pi would get better computing performance after offloading.

4.2.2 Tests on Empty Request

From the API prototype discussion of figure 3.1, round trip time(RTT) is the variable we concerned about within API usage since it consumes extra cost, excepting that stronger computing capability enables cost saving.

In the aim of investigating the impact of RTT as the essential cost for applying API on different servers, an easy communication test is proposed. In this test, Pi calls a special function named small_request in API library without any computation. When server receives this call, only an empty message transmitted back as the response. Thus, no computing time(CT) is needed and the fetched test execution time would be viewed as the whole transmission cost. The registered function is expressed below:

function small_request

```
1 def small_request():
2     return
```

The same test has been done, targeting both local and distant desktop, by calling the function small_request() 1000 times. As the RTT of each call is foreseeable to be short, the running time is collected in the unit of microsecond. All data is displayed in figure 4.4.

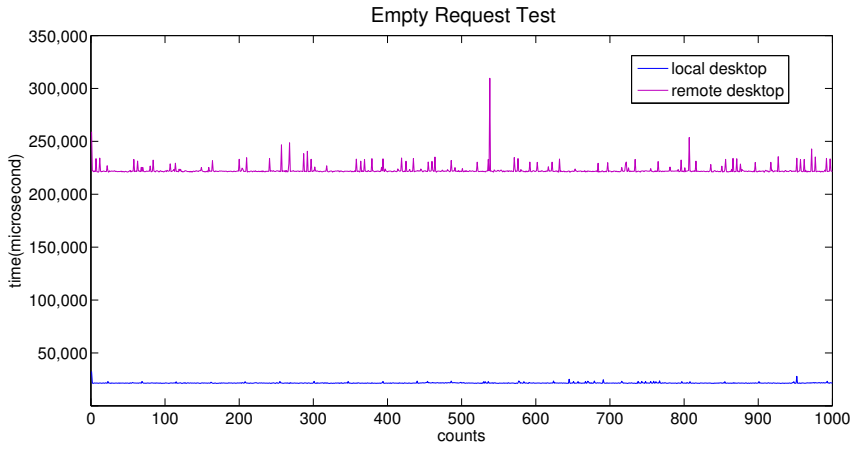


Figure 4.4: Empty request for different servers

The figure above states the basic API cost of local and remote desktop. Remote desktop cost number is more than 10 times greater, most of which is around 230000 microsecond, against most of those cost number for local desktop that reduces to 22000 microsecond. It indicates that the distance discrepancy leads to significant difference on API essential cost, and usually the local server is able to achieve better performance in the same hardware condition.

Descriptive results		
Metric	Local desktop	Remote desktop
Max:	32528	309686
Min:	21067	220814
Mean:	21574.49	222643.6
Median:	21479	221744.5
Mode:	21376	221869
Count:	1000	1000
Variance:	308408.66	18402450.2

Table 4.2: Statistical Results of Empty Request Test

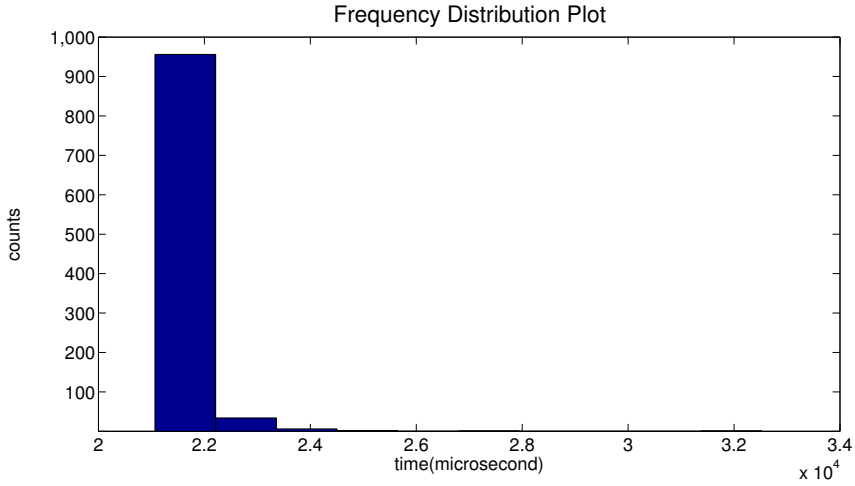


Figure 4.5: Empty Request Test on Local Desktop

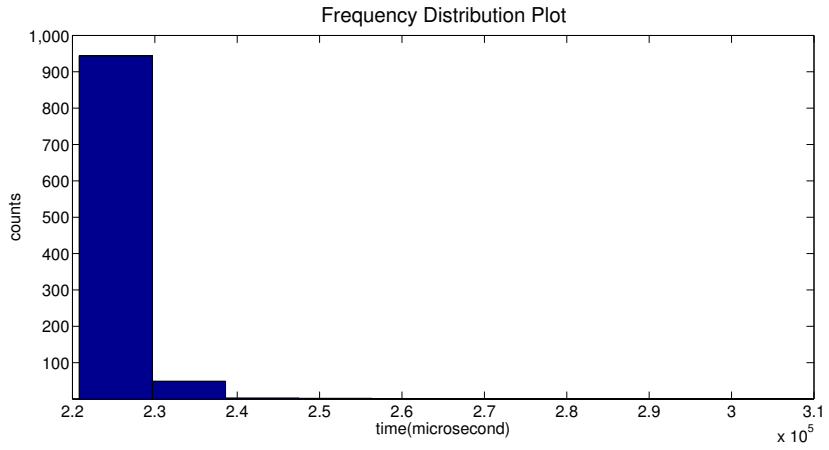


Figure 4.6: Empty Request Test on Remote Desktop

The data statistics table and frequency distribution plots presented above indicate that the data changes during 1000 calls. The maximum number retrieved from local desktop is 32528 while 309686 from remote one.

The similarity between these two numbers is that they are collected from the running time of first call, which shows that the first call costs more time than the following calls, as it demands connection initialization for building up the first communication. But the other calls could reuse this connection and enable spending less time as the interval period between each one is very short.

Table 4.2 indicates that the mean value is 21574.49 of local desktop, which increases 201069 microseconds against 222643.6 of remote as transmission cost. Since all the data from remote desktop is overall much greater than that from local one, which is consistent with the feature displayed in figure 4.4.

Another two frequency plots, 4.5 and 4.6, shown after table 4.2 display the data distribution in a more specific way. Both of them state that the data is distributed intensively and varies in small scale. Within 10% frequency distribution calculation behind, there are more than 95% of data locate around 22213 in local and about 94% of data are kept around 229701 in remote. These results demonstrate that our API is capable of setting up a relatively reliable connection for Pi and server communication since it achieves good stability even on remote desktop holding long distance.

4.2.3 Tests on Message Size

As discussed in the last chapter, the API prototype figure 3.1 shows that the request size(RS) would impact on RTT as well. In order to investigate how the variable RS affect API utility, we formulate two tests on changing transmitted message size while keeping the same computing complexity.

In these tests, Pi calls different registered functions in API library to get expected replies. The parameters passing to two functions are the same array. Both of these functions would process the received array. The computing operations are identical, which include counting the number of array element and then implementing array sorting. The only different behavior is that one of those two functions returns counting number back to Pi, while the other one would return the new sorted array which implies that the message size is larger than former one.

These two functions are explained as below:

different return functions

```
# return size of processed array
def return_array_quantity(data):
    a=len(data)
    sort_array=[]
    sort_array=sorted(data)
    return a

# return new array after sorting old array
def return_array(data):
    a=len(data)
    sort_array=[]
    sort_array=sorted(data)
    return sort_array
```

Functions are executed on both desktop and the execution time is collected as preference metric for performance evaluation. Results comparison figures are shown in the following:

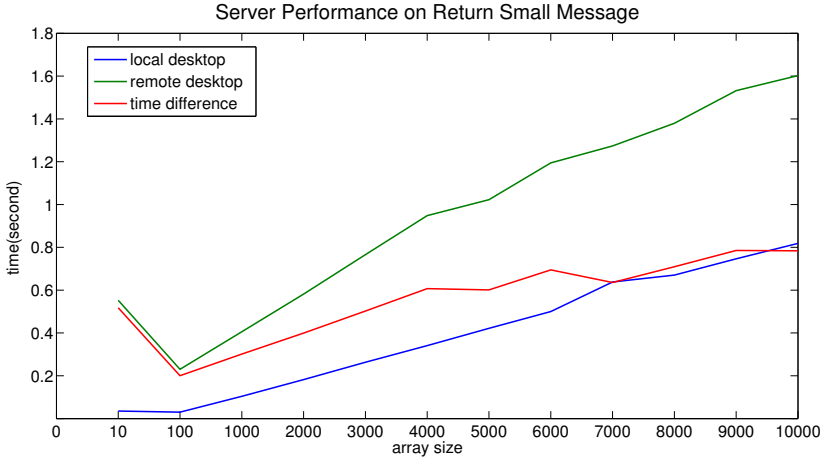


Figure 4.7: Test on small message

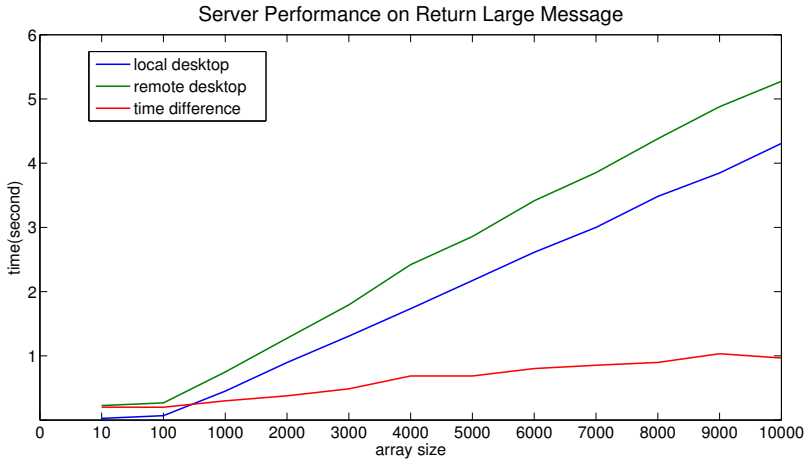


Figure 4.8: Test on large message

In two figures above, it makes sense that local desktop gets better performance all the time than remote one whatever transmitting small or large message in test, because remote desktop with long distance always demands more time on transmission. Therefore time differences of both pictures remain steady growth as transmitted array size getting bigger, but maintain in 1 second. In addition, the running time almost keeps linear growth in both tests as the array element number exceed 100.

However, only the result shown in first test of returning small message is not consistent with the rule concluded above, in which the running time of transmitting array with 10 elements is even longer than that with 100 elements. That is because of connection initiation as analyzed in last section.

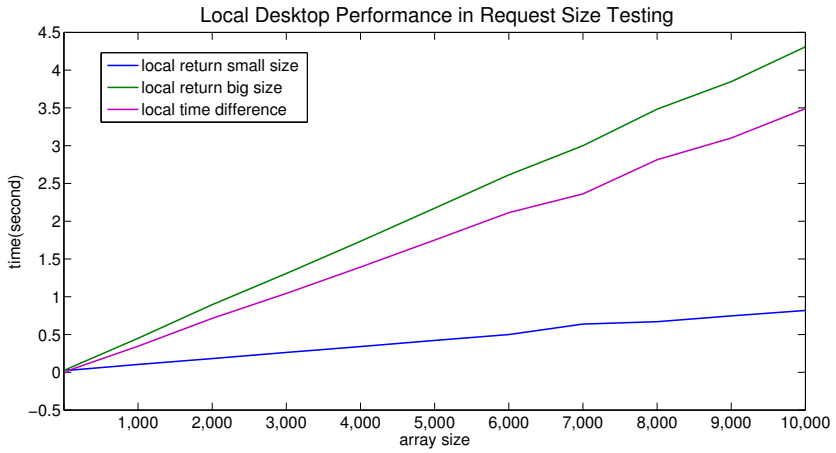


Figure 4.9: Performance on local desktop

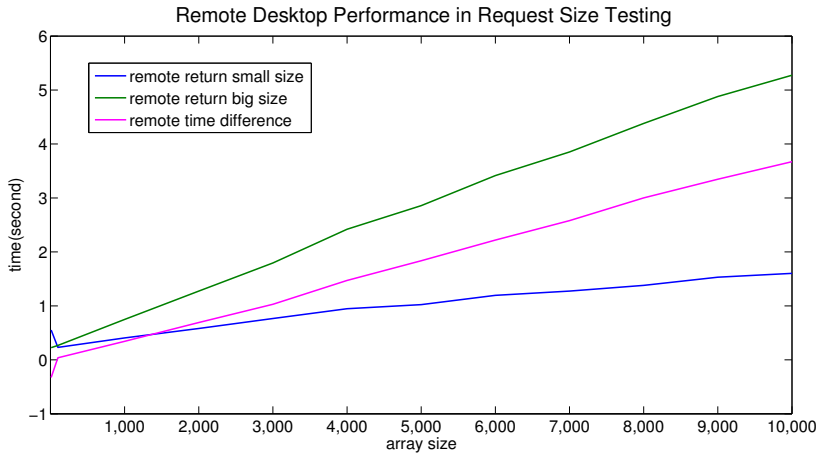


Figure 4.10: Performance on remote desktop

Figures 4.9 and 4.9 present data comparison of both tests in two different servers separately. They are very similar to each other shown in all results keeping linear growth as the array size getting increased, even the time variation keeps in a homogeneous way which is from about 0 to 3.5 second.

4.2.4 Tests on Increased Computing Complexity

Another variable in figure 3.1 named computing time(CT) is investigated in this test. Since the computing complexity determines CT, the test adopts a simple function named add_all_number() to increase complexity degree easily. As shown below, when given parameter getting bigger, the calculation process of this function is becoming more complicated which indicates that running time is expected to be longer.

function add_all_number

```
1 def add_all_number(n):
2     count=0.0
3     for i in range(1,n):
4         number=1.0/i
5         count+=number
6     return count
```

The test has been done on Pi, both with API and without API for 20 times, and all results are generated into figure 4.11 below. The computing counts could also be viewed as complexity level.

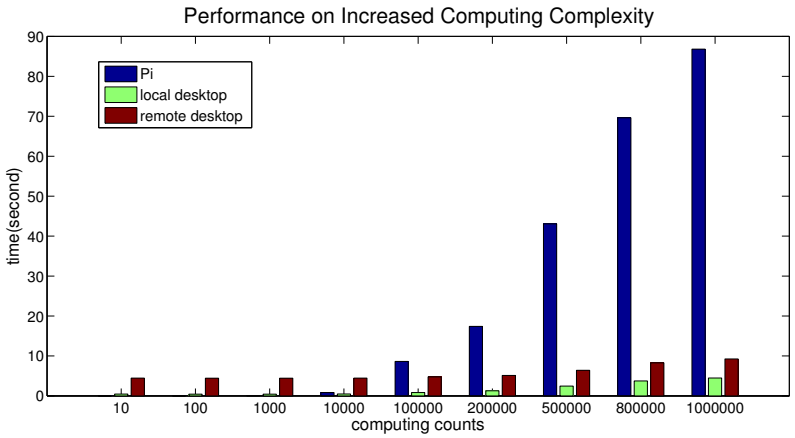


Figure 4.11: Performance on testing computing complexity

According to figure 4.11, the mechanism of offloading begins to show its advantage when the computing level raises to 100000. In addition to that, significant improvement occurs when computing level is higher than 200000. Meanwhile, the execution time reduces from 90 to 5 second at most achieved by local server when computing counts is as big as 1000000. It also illustrates that when computing complexity grows, Pi may reveal its weakness more clearly.

Since the computing performances of two servers appear to be much better than Pi, and the variation between local and remote desktop is hard to be perceived in figure 4.11, those two figures below extract servers performance data and display them specifically.

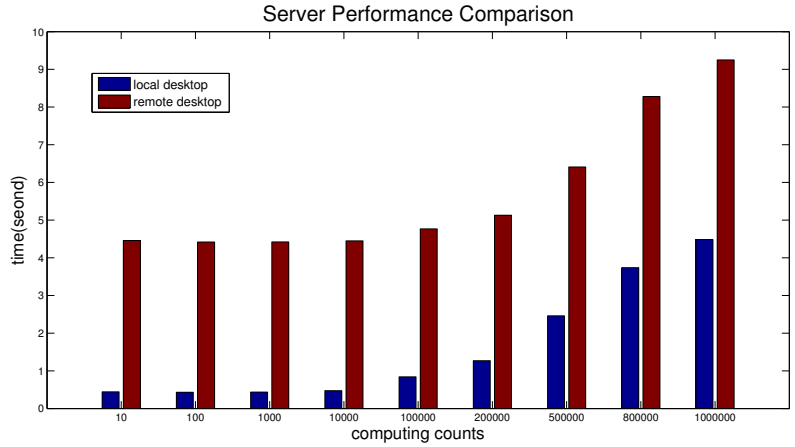


Figure 4.12: Performance comparison between local and remote desktop

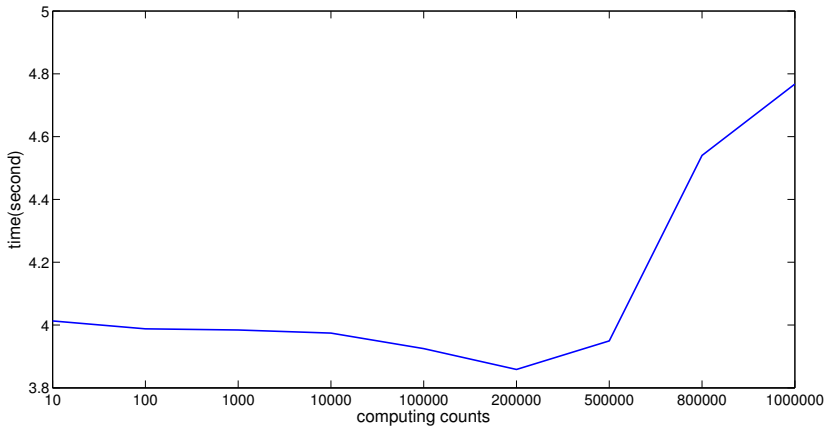


Figure 4.13: Server Performance Variation

Figure 4.12 illustrates that when the computing complexity of request stays in same level, local desktop always has better performance on processing request than remote one, which is consistent with the result shown in section 4.2.3. It also shows that when computing counts is less than 10000, the time difference between local and remote remain in similar values, which is about 4 second presented in figure 4.13. Moreover, the variation reduces slightly from counts 100000 to 500000, then keeps increasing until as long as 4.75 seconds at last with computing count number 1000000.

In order to make the comparison of all results in detail, some calculations are conducted to observe the performance improvement rate of API utility. The formula could be expressed as below and the improvement rates of both local and remote desktop against various complexity degrees are displayed in table 4.3.

$$local\_increase\_rate = \left(\frac{Pi\_performance - local\_performance}{Pi\_performance} \right) \times 100\% \tag{4.1}$$

$$remote\_increase\_rate = \left(\frac{Pi\_performance - remote\_performance}{Pi\_performance} \right) \times 100\% \tag{4.2}$$

Computing Counts	Local desktop	Remote desktop
10	-1750%	-18475%
100	-163.4%	-2595.1%
1000	72%	-182.4%
10000	97.2%	73.5%
100000	99.5%	97.2%
200000	99.6%	98.5%
500000	99.7%	99.3%
800000	99.7%	99.4%
1000000	99.7%	99.5%

Table 4.3: API Improvement Rate on Increase Computing Test

The table 4.3 above explains how much improvement the API achieves against without applying offloading, where all operations are implemented on Pi locally. It is clear that positive improvement rate 72% begins from computing degree 1000 when offloading to local desktop, while for remote desktop, it is 73.5% appeared until degree 10000 since it demands more essential cost because of longer distance. Besides, as computing complexity level grows, the improvement rate of local desktop keeps increasing as well, then remains in a relative stable value of 99.7%, while the remote one grows from 73.5% to 99.5% in the end of test.

There are also some rates shown as negative value in table 4.3. Because transmission cost of servers accounts most of the total running time, but it still spends less time than computing on Pi. Thus, result explains the API could only get positive performance when computing degree is higher than 1000 for local desktop and 10000 for remote desktop.

4.2.5 Tests on Sorting Algorithms

The API we built would mainly focus on educational ICT environment, and the more precise purpose is for basic computer science teaching. In order to adjust our tests more close to practical situation on programming teaching, two common cases that is about sort algorithms programming are proposed in our experiments.

We select bubble sort and merge sort as our sorting algorithm examples, not only because of their popular usage in programming teaching, but also for their discrepancy in sorting ability. These two algorithms could be explained in pseudo-code as below, and the actual python code used in following tests have been implemented in paper [14].

bubble sorting

```
# assume A has n elements
bubblesort(A):
  for i=1 to n-1:
    for j=0 to n-i:
       compare A[j] and A[j+1]
       if A[j] > A[j+1]:
          A[j],A[j+1] = A[j+1],A[j]
```

merge sorting

```
# assume A has N elements
mergesort(A):
  #step1:
  split A into N arrays, and n=N
  #step2:
  sorting and merging n arrays in neighbouring pairs,
  and n/2 new arrays are produced;
  #step3:
  n=n/2;
  repeat the second step until n=1;
```

Just as described above, both of these two algorithms are easy to understand and not hard to be implemented by programming. Generally, bubble sorting has higher time complexity than merge sorting since it demands more comparison processes. The time complexity is shown below:

$$BubbleSorting : time\_complexity = \bigcirc(n \wedge 2)$$
$$MergeSorting : time\_complexity = \bigcirc(n \log n)$$

We test these algorithms targeting both Pi locally as well as two servers for 20 times. The arrays sorted in the test are produced by random numbers and the size of arrays grows from 10 to 5000. Besides, running time is generated from each test where two algorithms would process the same array in order to guarantee data validity.

However, when the baseline test was running on Pi, some abnormal situations happened which are out of expectation. These cases are described as below and corresponding results are shown separately.

1 Running baseline test on bubble and merge sorting in one script, and both algorithms process the same array generated by 5000 random numbers.

Sorting Algorithm	time(second)
bubble	118.351107
merge	0.862692

Table 4.4: Case 1 on Pi

2 Running baseline test on only bubble sorting in one script, and it processes one array generated by 5000 random numbers.

Sorting Algorithm	time(second)
bubble	108.38145

Table 4.5: Case 2 on Pi

3 Running baseline test on only merge sorting in one script, and it processes one array generated by 5000 random numbers.

Sorting Algorithm	time(second)
merge	4.589558

Table 4.6: Case 3 on Pi

4 Running baseline test on bubble and merge sorting in one script, but two algorithms process different arrays that generated by 5000 random numbers separately.

Sorting Algorithm	time(second)
bubble	102.148386
merge	4.051953

Table 4.7: Case 4 on Pi

From these tables shown above, huge differences appear between case 1 and 3, in which the merge sorting time increases from 0.863 to 4.59 seconds but bubble sorting keeps relatively stable results from 118.35 to 108.4 seconds in case 1 and 2. Since every array is generated by random numbers that results in system error on sorting time, it makes sense that variation happened for bubble sorting but it is unacceptable for merge sorting.

The case 4 shows more reliable results than case 1 when it is compared with case 2 and 3, that is because it supports processing different arrays for two algorithms. As merge sorting is executed after bubble sorting in script, the sorted array processed by bubble might be maintained in CPU. Hence when merge sorting asks for processing the same array, Pi just returns memorized sorted array directly which results in extremely short execution time for merge sorting test. Those results shown in all 4 cases just demonstrate that.

Apart from Pi, whether these two servers we adopted have memorability in processing request, would affect the following test as well. In order to get the answer in a relative short time, we select array size 100 and 300 for testing. The results on local desktop are displayed in table 4.8 below:

Testing Cases	Size	Bubble(s)	Merge(s)
1.process the same array,running 1 time	500	0.302371	0.252606
	1000	0.660866	0.489824
2.prcess the same array,running 20 times	500	6.046051	5.163192
	1000	13.128396	9.768153
3.process different arrays,running 1 time	500	0.311585	0.254523
	1000	0.671638	0.482085
4.process different arrays,running 20 times	500	5.872934	5.048051
	1000	13.188229	9.600002

Table 4.8: Server memorability testing

Table 4.8 illustrates that there is no memorability on servers when they process requests from API. The tables, case 1 and 3, show relatively similar results on sorting array, which indicate that computing data of former request do not affect computing process for following one. When asked for

sorting the same array, the server will execute them individually without computing overlap. Moreover, results of cases 2 and 4 provide more support for previous conclusion. They imply that after running test for 20 times, those data increasing still keep similar relatively, and both increase approximately as much as 20 times than former tests of cases 1 and 3 respectively.

Since we get the conclusion that sorting same array within two algorithms in one script, will affect execution time on Pi, the following baseline and API utility tests are running separately in different scripts.

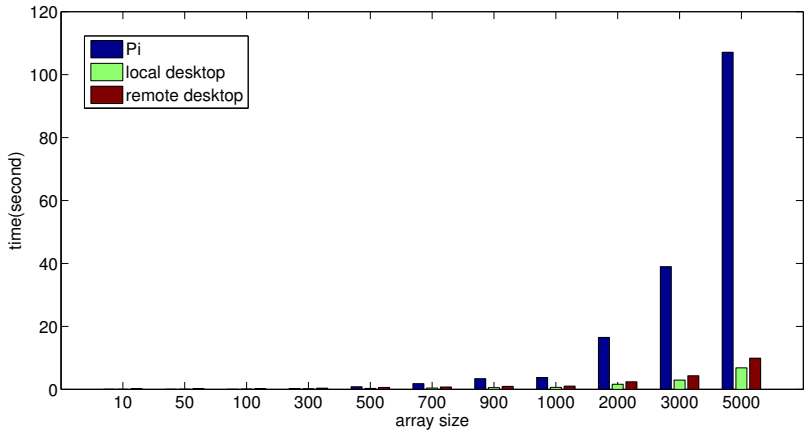


Figure 4.14: Bubble sorting algorithm test

Figure 4.14 displays results of bubble sorting test. It illustrates that the API enables increased improvement in processing array with bubble sorting algorithm as array size getting bigger, especially when it exceeds 2000. In this figure, servers performances appear to be similar, since compared with Pi, they all achieve obvious progress. In order to show servers discrepancy in performance more specifically, figure 4.15 is generated to show below:

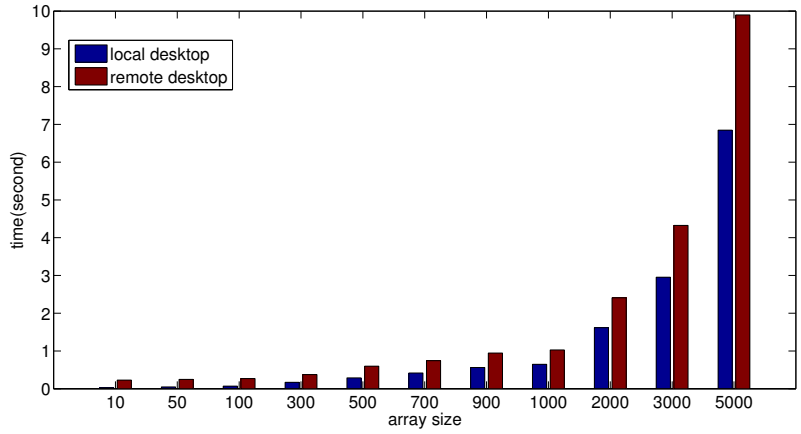


Figure 4.15: Server comparison of Bubble sorting algorithm test

Same as those tests before, local desktop achieves better performance than remote one through all numbers. Meanwhile, the difference also keeps increasing as array size grows, because larger array costs more transmission time and cause more instability on the way.

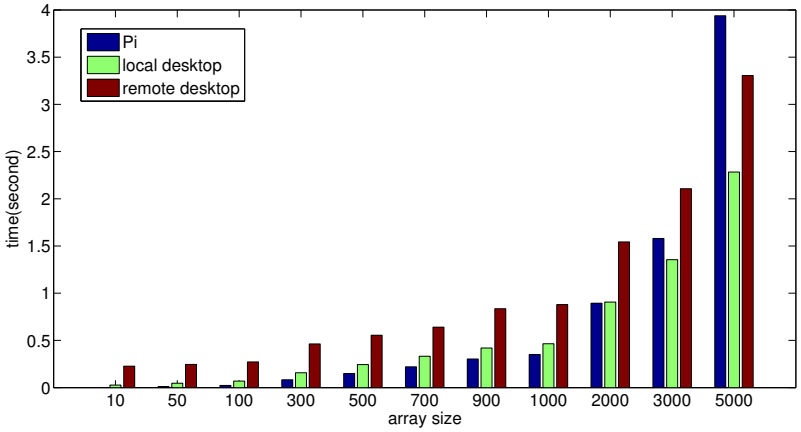


Figure 4.16: Merge sorting algorithm test

Figure 4.16 above displays results of merge sorting test. It indicates the API leads to worse performance when array size is smaller than 2000 on local desktop and 5000 on remote respectively. Since merge sorting within lower time complexity meaning that the sorting process would be faster on all devices, it is viewed as an optimized algorithm other than bubble and it enables spending less time but demands more swap space. Therefore, it makes sense that merge sorting performs not as effectively as it shown in figure 4.14.

Size	Pi_bubble	Pi_merge	Local_bubble	Local_merge	Distant_bubble	Distant_merge
10	0.000405	0.001777	0.0271088	0.02654165	0.22782395	0.22660715
50	0.008661	0.011009	0.0471825	0.04603805	0.24675575	0.24587555
100	0.028082	0.023434	0.06981695	0.06864815	0.2697824	0.2716662
300	0.274903	0.083003	0.17126725	0.15690855	0.37356485	0.4630151
500	0.825981	0.148802	0.28805675	0.24391015	0.59839715	0.554569
700	1.828227	0.220020	0.416831	0.3322628	0.74332155	0.63994045
900	3.385254	0.303188	0.56143775	0.4191557	0.9453837	0.8356844
1000	3.768765	0.351251	0.6465681	0.4649777	1.02860055	0.8786762
2000	16.486658	0.893767	1.61751775	0.9059603	2.40922505	1.5425208
3000	39.002199	1.579592	2.9532787	1.3554098	4.32415625	2.10634985
5000	107.115410	3.938226	6.84696965	2.28369495	9.8972693	3.3053919

Table 4.9: All performance data from sorting algorithm tests

In the aim of having better understanding of performance data, table 4.9 above displays all results with different colors assigned for the first positive performance value on local and remote desktop.

In general, bubble sorting achieves greater progress shown in table 4.9 that as array size getting bigger, the improvement increases from 0.1 second against size 300 to 100 seconds against size 5000, while for remote desktop, it increases from 0.2 second in the size of 500 to 97 seconds in the size of 5000.

However, in merge sorting performance comparison, table 4.9 indicates local desktop achieves positive improvement value until array size raises to 3000 within 0.22 second, while remote one gets 0.6 second better until size grows to 5000.

As formulas listed in section 4.2.4, improvement rate of each test is calculated to describe that applying API enables how much efficiency targeting local and remote desktop separately. All rates are presented in following two figures, and the marked colors are also consistent with those in table 4.9.

Array Size	Local desktop	Remote desktop
10	-6675%	-56850%
50	-442.5%	-2727.6%
100	-148.4%	-860.1%
300	37.7%	-35.9%
500	65.1%	27.6%
700	77.2%	59.3%
900	83.4%	72.1%
1000	82.8%	72.7%
2000	90.2%	85.4%
3000	92.4%	88.9%
5000	93.6%	90.8%

Table 4.10: API Improvement in Bubble Sorting Test

Array Size	Local desktop	Remote desktop
10	-1372.2%	-12488.9%
50	-318.2%	-2135.5%
100	-193.2%	-1061.1%
300	-89%	-457.8%
500	-63%	-272.7%
700	-51%	-190.9%
900	-38%	-175.6%
1000	-32.4%	-150.1%
2000	-1.4%	-72.6%
3000	14.2%	-33.3%
5000	42%	16.1%

Table 4.11: API Improvement in Merge Sorting Test

Improvement rates in tables 4.10 and 4.11, illustrate that API could get positive performance until the array size raises to a certain number, which means computing complexity reaches to a specific extent. The positive performance occur in the array size of 300 with rate 37.7% and in the size of 500 with 27.6% on bubble sorting for local and remote desktop separately, while in the size of 3000 within 14.2% and 5000 within 16.1% on merge sorting.

Negative rates appear as well especially in merge sorting, and it accounts the most of rate values for remote desktop, where it keeps worse performance until in last test with size 5000. But as size number grows, which implies increased computing complexity, negative rate values are approaching closer to 0.

Thus for merge sorting, the overall result is not desirable for using API as it leads to worse performance when sorted array size is smaller than 2000. It inspires us to develop a more smart API for realizing adaptive per-

formance to meet different needs of requests.

4.3 Process Monitoring

When those tests described from section 4.2.2 to 4.2.5 are running, a monitor script is executed on Raspberry Pi periodically for monitoring the process performance. There are 8 different metrics as shown in table 3.2 are chosen to collect into data files.And those data files by offloading to local desktop are plot as following figures.

The explanation of legend in figures is in the table 4.12 below:

Legend	Description
small test	empty request test
size test	various message sizes test
computing test	increased computing complexity test
sorting test	sorting algorithms test

Table 4.12: Figure legend description

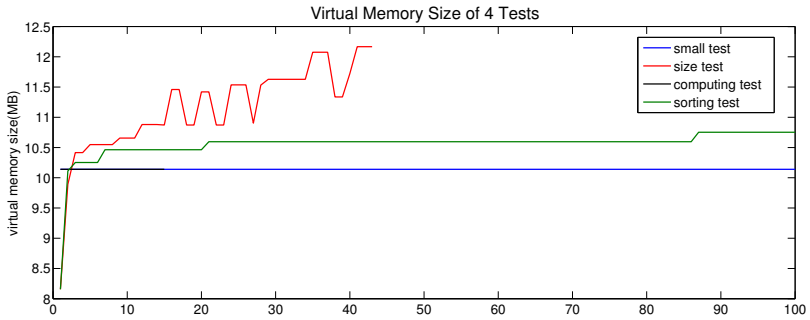


Figure 4.17: /proc/$pid/status: Vmsize

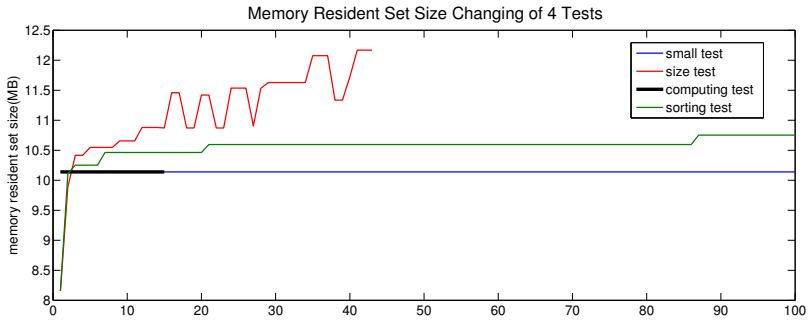


Figure 4.18: /proc/$pid/status: VmRSS

Those two figures above show allocated memory information of testing process. The virtual memory size changing of these 4 tests in figure 4.17 is almost the same as real memory size variation in figure 4.18. That is because these 4 tests are more related to math computing process, which has more influence on CPU occupation rather than memory allocation.

Since all tests are not memory consuming and there is no need to use swap space, memory allocated in virtual memory for these processes is all locating in real memory as well. Thus two figures above look like exactly the same.

In addition, size and sorting test consume more memory than other two tests shown in figures. The reason of that is size and sorting test demand to generate random arrays, and as array size growing, the process needs more space to store arrays for following transmission or sorting procedure. Thus the memory size assigned for these two tests are larger.

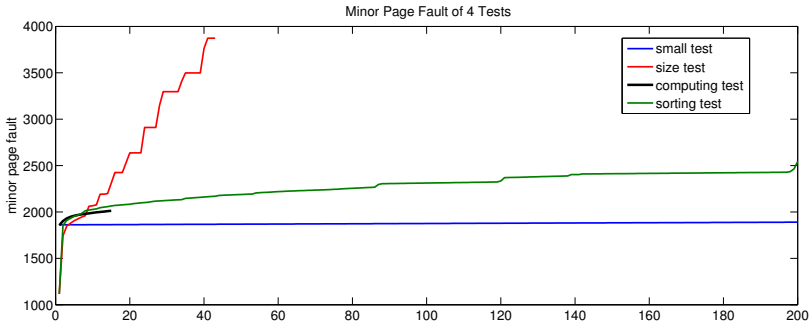


Figure 4.19: /proc/$pid/stat: min_flt

The figure 4.19 above shows the number of minor page fault made by process. It indicates page fault occurs but process does not need to load a memory page from disk. Thus there is no big time latency caused by disk operation for process performance.

In those data files, all numbers of major fault are shown as 0. Since the memory allocated for process is all in real memory, thus there is no possibility for it to load memory page from disk.

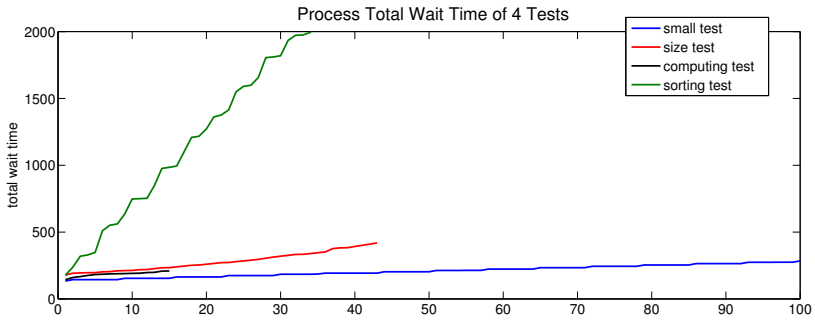


Figure 4.20: /proc/$pid/sched: wait_sum

Since the process execution time is influenced by offloading performance and the performance is discussed in detail in last sections, so only the wait time of process is plot above. Figure 4.20 illustrates that the sorting test process is kept in wait state for longest time, even longer than size test. That is because both transmission and sorting procedures cost more time than other 3 tests, as it needs to transmit large array forth and back every time, in addition, adopting less effective algorithm to process sorting.

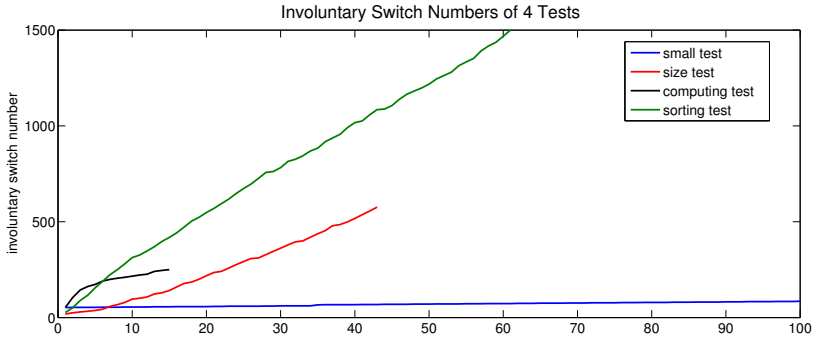


Figure 4.21: /proc/$pid/sched: nr_involuntary_switches

The file '/proc/$pid/sched' stores CPU scheduling information for process. And the metric 'nr_involuntary_switches' indicates how many times the process is forced to take off CPU. This is because the process has exhausted CPU time slice and kernel would switch it to grant CPU time to other processes. The sorting test still gets the biggest involuntary switch times, as offloading array sorting to server requires more waiting time on array transmission.

Those monitoring results by offloading to remote desktop are very similar to local one, so all generated figures are shown in Appendix.

4.4 API Optimization

From the description and comparison of all tests data in last section, it is apparent that our API still has its limitation until now. For instance, when processing array with merge sorting algorithm, API could only achieve positive improvement until array size is larger than 5000. It is hard to say that in practical situation, whether students are required to test this with array having large number size. Hence it is unpredictable that the API enables increased performance.

Improving API to realize more efficiency under different conditions is what we pursue in this section. Since API within offloading all requests could not always achieve our goal, within no offloading is added into API strategy as well. It implies there is threshold stored in API that determines which policies will be deployed, offloading to server or executing locally, named smart offloading.

Thus the mechanism adopted by our new smart API includes two features rather than offloading all the time. The next issue is finding the threshold value. For those two sorting algorithm tested in section 4.2.5 above, threshold values are explained as arrays size in figures 4.10 and 4.11 obviously. For bubble sorting, threshold on local desktop is the size between 100 and 300, while on remote desktop it is between 300 and 500. For merge sorting, the threshold value is between 2000 and 3000 on local desktop and between 3000 and 5000 on remote one.

4.4.1 Pre-test Trials

Since threshold value scope is guessed as above and in order to test the feasibility of smart offloading, we create a file stored threshold values at first and set them as 300 and 3000. Then a Python module named Jun is initialized for prejudging computational offloading or not according to values listed in threshold file.

This means before processing sorting, array size requires to be compared with threshold value. If it is smaller than 300 on bubble sorting or smaller than 3000 on merge sorting, the sorting process will be handled by Pi locally, otherwise it will be offloaded to server and wait for result. The basic policy is expressed easily below:

Smart_API Policy

```
1 A=array size
2 B=threshold value
3 if A<B: process sorting locally
4   else: process sorting by offloading to server
```

Experiment is still tested on sorting algorithms for 20 times and results

of bubble sorting is presented as fllowing two figures:

```
root@raspberrypi:~/scripts# ./initial_smart.py
----------------------------------------
Smart Offloading test for bubble sorting
Array size:100
Without offloading:     0.727742
All offloading:         1.549298
Smart offloading:       0.836381
----------------------------------------
```

Figure 4.22: Smart offloading on bubble 100

```
root@raspberrypi:~/scripts# ./initial_smart.py
----------------------------------------
Smart Offloading test for bubble sorting
Array size:300
Without offloading:     6.281761
All offloading:         3.274839
Smart offloading:       3.353994
----------------------------------------
```

Figure 4.23: Smart offloading on bubble 300

Figures 4.22 and 4.23 illustrate that smart offloading is successful on making offloading decision. The first figure is tested on array size 100, which is smaller than threshold value 300, and it indicates the sorting performance will be better on processing locally. Then results number 0.727742 and 1.549298 demonstrate that. For smart offloading, it checked and compared the array size 100 with threshold before every test and then decided to process sorting on Pi. The final running time 0.836381 seconds is closer to number 0.727742, but not exactly the same because the mechanism of without offloading is executing array sorting on Pi directly without any estimation.

The second figure 4.23 is tested on array size 300, which is equal to threshold and demands to deploy offloading according to API policy. The results meet our expectation that offloading achieves better performance as 3.274839 against 6.281761 seconds. For smart offloading, the final running time 3.353994 seconds is closer to mechanism of all offloading this time and the slight variation is within the same reason of estimation cost.

Therefore, the smart API is capable of making more effective decision on adopting offloading and results on merge sorting below support this conclusion as well:

```
root@raspberrypi:~/scripts# ./initial_smart.py
----------------------------------------
Smart Offloading test for merge sorting
Array size:2000
Without offloading:     18.806552
All offloading:         19.145634
Smart offloading:       18.87361
----------------------------------------
```

Figure 4.24: Smart offloading on merge 2000

```
root@raspberrypi:~/scripts# ./initial_smart.py
----------------------------------------
Smart Offloading test for merge sorting
Array size:3000
Without offloading:     36.751364
All offloading:         28.574907
Smart offloading:       28.555234
----------------------------------------
```

Figure 4.25: Smart offloading on merge 3000

Two figures above show the smart API realizes better performance rather than all offloading in the first one and without offloading in the second one. But in figure 4.25, the running time of smart offloading is even slightly shorter than all offloading mechanism which is not consistent with the analysis that smart would cost extra time on comparison. However, taking network latency and test repeat times into consideration, the error is in an acceptable scope and supposedly caused by accumulated networking condition discrepancy of each test.

4.4.2 Preliminary Threshold Building

Since the module Jun has already achieved making smart offloading decision based on threshold file, next step is producing this threshold file with higher precision automatically rather than guessing from data file.

An initial function is supposed to be included in the module for creating threshold file. There are two parameters that are required to be passed into module for threshold file establishment. The first one is the host name of external server as offloading target. The second one is accuracy interval which describes array size in our experiment. And that indicates each test would sort array with the growth size of interval number at a time. Thus the module of API would call initial function to run when this module is be executed as below:

./Jun.py 'server_hostname' interval_number

Given those two arguments explained above, the initial function will

begin to execute threshold-building tests. The pseudo-code is interpreted as following:

function init()

```
$start_size=0;
bubble=merge=true;

while bubble or merge:
  $start_size+=interval
  # start to test threshold value of bubble sorting
  at array size $start_size
  if bubble:
     generate random array with size $start_size;
     sorting array locally and recording running time as t1;
     sorting array by offloading to server,
     and recording running time as t2;
  # compare execution time t1 and t2
     if t1<=t2: bubble=true;
     else: bubble=false;
           $threshold_bubble=$start_size;
           write $threshold_bubble into threshold file;

  # start to test threshold value of merge sorting
  at array size $start_size
  if merge:
     generate random array with size $start_size;
     sorting array locally and recording running time as t1;
     sorting array by offloading to server,
     and recording running time as t2;
     if t1<=t2: merge=true;
     else: bubble=false;
           $threshold_merge=$start_size;
           write $threshold_merge into threshold file;
```

These codes above explain the basic logic of how to establish threshold file. Two boolean variables 'bubble' and 'merge' indicate threshold finding status for both algorithms, and they are initialized as 'true' that means threshold value has not been found yet. As array size growing, the powerful computing capacity of server will make up transmission cost in total running time, hence server overall performance will catch up and then surpass Pi. Meanwhile, the boolean variable is modified into 'false', implying threshold value is achieved and no more tests on this algorithm is needed.

In addition, it is not credible enough to adopt threshold values got from one test comparison, thus sorting algorithms would be executed for 20 times and the total running time is used for comparison. However, as discussed in 4.4.1, reprocessing the same array would affect running time on

Pi and cause inaccuracy result comparison, and generating 20 different arrays for sorting in each test consumes much extra time, so the performance on Pi will choose the value as big as 20 times of first test result.

Considering the remote desktop would spend more time on transmission which may lead to more instability for result, interval values 100 and 50 are chosen for remote and local desktop separately. Thus these threshold values selected by both local and remote desktop after running initial tests are displayed as following figures:

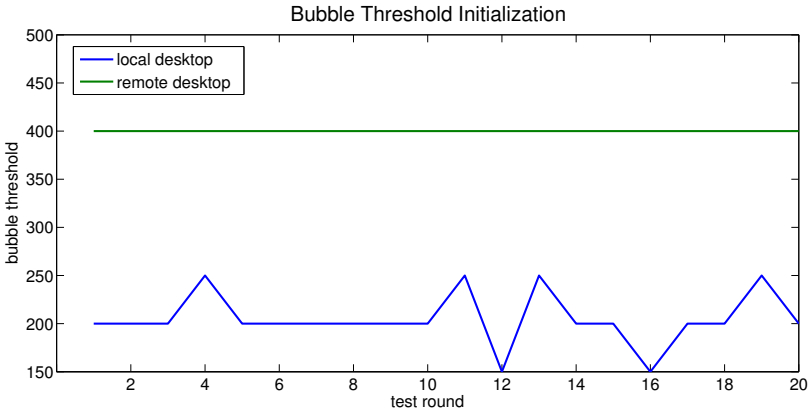


Figure 4.26: Initialize threshold value for bubble sorting

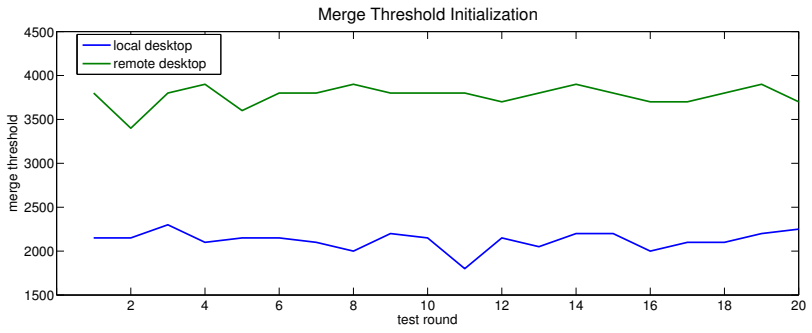


Figure 4.27: Initialize threshold value for merge sorting

Figure 4.26 shows threshold values of bubble sorting are consistent with our tests before in 4.2.5, which are around 200 and 400. Since bigger interval value is selected for remote desktop testing, and bubble sorting is not an effective algorithm that would easily cause intensive computing for Pi within small array size, so remote desktop could display a more stable threshold than local one.

When array size getting bigger to thousands of elements, the variation is more obvious in figure 4.27. Because merge sorting is an optimal algorithm with lower level of computing complexity. So the Pi could deal

with that until size grows to a big number but larger size implies more transmission cost, which may affect final running time. The statistical information is presented below:

Variable	local bubble	remote bubble	local merge	remote merge
Mean	205	400	2125	3700
Std	27.6253	0	106.9924	117.4286

Table 4.13: Threshold Initialization Statistics

4.4.3 Smart Offloading

Those mean threshold values shown in table 4.13 are adopted to do smart offloading tests for bubble and merge sorting on local and remote desktop separately. Those results are displayed in following figures:

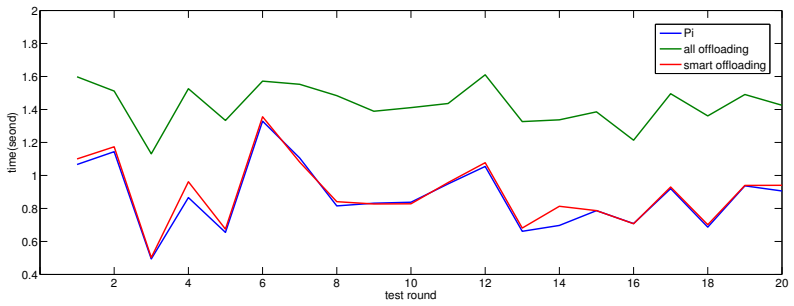


Figure 4.28: Smart offloading bubble to local desktop

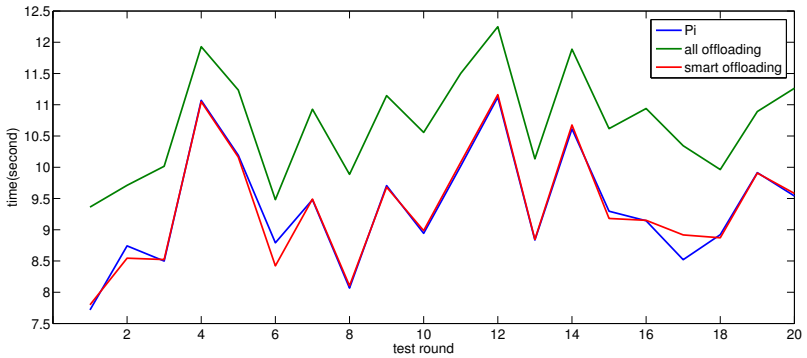


Figure 4.29: Smart offloading merge to local desktop

Two figures above are bubble and merge sorting results by smart offloading to local desktop with threshold values 205 and 2125 separately. In order to show the performance discrepancy between all offloading and

smart offloading more obviously, random array generation of those tests is in the limit size of 200 for bubble, and 2000 for merge. It indicates all produced random arrays would get better performance to execute locally.

Figures 4.28 and 4.29 exactly illustrate our prediction. All performance of smart offloading turns out to be more effective than that of all offloading. And they are very close to the performance of no offloading. The slight variation between smart offloading and running on Pi is because smart offloading needs extra deterministic process.

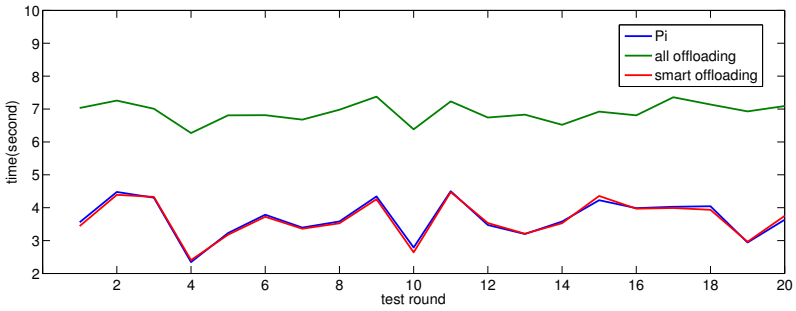


Figure 4.30: Smart offloading bubble to remote desktop

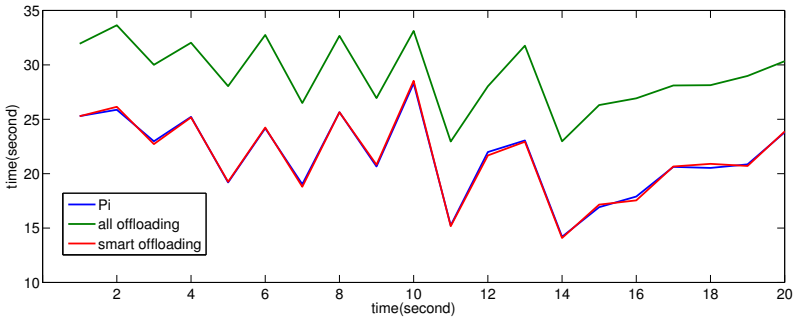


Figure 4.31: Smart offloading merge to remote desktop

These two figures are bubble and merge sorting results by smart offloading to remote desktop with threshold values 400 and 3700 separately. With the same reason of easily showing performance discrepancy, these size limitations for generating random arrays are 400 for bubble and 3700 for merge on remote desktop.

As shown in figures 4.30 and 4.31, they illustrate similar variation results as those on local desktop except for value range. Remote desktop always demands more time for processing offloading, as extra time are spent on the transmission way. That is also the reason that performance discrepancy between smart offloading and no offloading is even smaller than that on local desktop, which is displayed in figures that those two lines are al-

most overlapped. It indicates the time spent on deterministic process has quite few impact on total running time relatively.

The figure 4.32 below presents various performance increase rate got by smart offloading compared to all offloading. These rates show bubble sorting always achieve more stable improvement rate than merge sorting, as the size range of generated array for merge sorting is much wider than bubble, and it means the variation of size numbers would influence total running time with a greater likelihood.

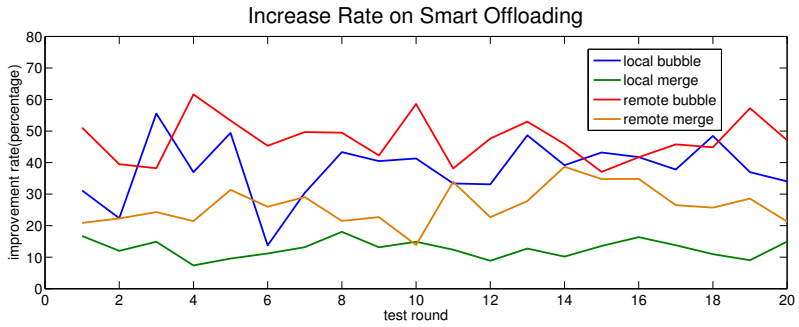


Figure 4.32: Smart offloading Increase Rate Comparison

The following table 4.14 is the statistical information of performance increase rates. It implies offloading to remote desktop always achieves higher increase rate than that to local desktop, as smart offloading to remote server would save more time spent on transmission.

Variable	local bubble	local merge	remote bubble	remote merge
Mean	38.060%	12.685%	47.372%	26.405%
Std	9.600%	2.837%	6.990%	6.066%

Table 4.14: Increase rate statistics

Since there is size limit for generating random arrays in former tests and they are all smaller than threshold values, following tests are going to demonstrate that smart offloading could also realize effective performance in the situation, where the size of random array may exceed threshold values. They are set in the range (0,1000) for bubble sorting and (0,10000) for merge sorting with offloading both to local and remote desktop.

Following two tables present overall improvement rates in 20 tests and each test collects the total running time of processing array sorting 20 times in different mechanisms. Variables 'bubble_rate' and 'merge_rate' indicate increase rates got from smart offloading compared to no offloading, while variables 'bubble_re_rate' and 'merge_re_rate' display improvement rates

of comparison between smart offloading and all offloading.

bubble_rate	bubble_re_rate	merge_rate	merge_re_rate
77.688%	-0.634%	63.969%	-1.630%
79.809%	1.922%	62.169%	-0.736%
77.053%	0.469%	62.060%	-1.544%
77.841%	0.639%	64.077%	-0.028%
81.128%	-0.487%	66.516%	1.057%
79.682%	0.759%	56.776%	0.013%
74.403%	1.817%	68.123%	-0.572%
75.999%	0.009%	62.204%	-1.060%
79.328%	0.640%	62.440%	-0.404%
77.854%	0.091%	65.407%	-1.382%
79.301%	0.322%	64.148%	-0.041%
80.536%	0.796%	63.419%	0.300%
71.366%	3.256%	60.027%	0.523%
82.295%	-0.488%	64.033%	-1.146%
79.347%	-1.294%	65.647%	-1.010%
81.849%	2.058%	64.005%	-0.627%
77.904%	1.018%	54.849%	-0.238%
78.388%	0.175%	59.164%	0.582%
74.910%	0.894%	57.781%	-1.121%
79.087%	0.452%	66.221%	-1.113%

Table 4.15: smart offloading to local desktop

The table 4.15 shown above illustrate smart offloading enables much more efficiency on performance than no offloading, as most of these increase rates are above 60%. While those relative increase rates compared with all offloading is not as obvious as that shown in figure 4.32, and several of them are even negative values. That's because the growing of size range set leads to increased possibility of generating random array with a larger size that threshold value, so offloading is always the better choice. And smart offloading needs prejudge cost which may result in negative increase rate when compared to all offloading.

bubble_rate	bubble_re_rate	merge_rate	merge_re_rate
51.515%	2.016%	53.602%	-0.276%
58.008%	6.854%	40.948%	2.421%
61.174%	14.195%	49.414%	1.331%
66.163%	8.470%	50.295%	1.090%
60.570%	7.216%	49.142%	3.986%
59.791%	11.260%	54.700%	3.544%
53.280%	4.437%	47.292%	3.476%
54.838%	15.628%	41.283%	-0.093%
57.776%	11.392%	47.801%	2.829%
48.787%	9.093%	60.054%	0.946%
65.900%	6.713%	47.469%	0.497%
61.229%	13.023%	53.597%	-0.195%
65.047%	5.596%	47.190%	2.055%
48.295%	15.334%	57.184%	-1.960%
55.982%	9.571%	53.405%	3.984%
54.576%	12.008%	53.629%	5.169%
53.993%	10.549%	49.170%	2.595%
61.529%	13.208%	54.399%	1.682%
64.374%	5.274%	45.045%	6.337%
65.829%	6.193%	50.764%	0.874%

Table 4.16: smart offloading to remote desktop

Smart offloading to remote desktop presents similar situations as that to local one, as the increase rate on no offloading is more apparent than all offloading. But from the comparison between tables 4.16 and 4.15, it is easily seen that remote desktop achieves higher increase relative rates to all offloading, as there are rare negative values and the greatest one is over 15%.

There is discrepancy between two tables, because smart offloading is capable of saving more time than all offloading when the prejudge process chooses no offloading. And transmission cost is not ignorable in the table.

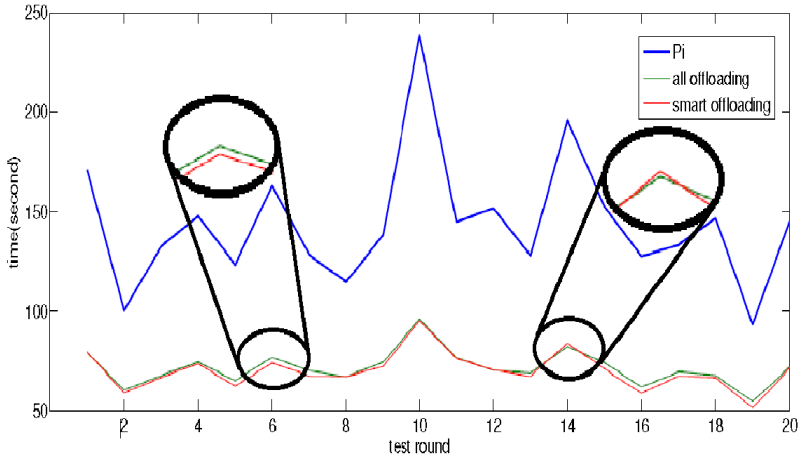


Figure 4.33: How smart offloading performs

Figure 4.33 above describes how smart offloading performs in testing merge sorting by offloading to remote server. Two subpictures inside figure 4.33 magnify the performance variation between merely offloading and smart offloading. They make it clear that compared with all offloading, there are both negative and positive performance of smart offloading. Since each test round would process merge sorting for different arrays 20 times, hence the magnification inset on left side showing the positive variation in test round number 6, indicates the accumulated performance of smart offloading is more effective than no offloading.

But the inset on the right side shows negative variation in test round number 14, where accumulated running time of smart offloading is greater than that of all offloading. It states that most of the 20 requests demands to be offloaded, and smart offloading requires a prejudge process, hence merely offloading saves more time and shows higher efficiency.

From all results of smart offloading shown above, it illustrates smart offloading could get overall more efficient performance when compared with no offloading and all offloading. Although it still leads to decreased performance some times compared with all offloading, those negative values are very small and could be accepted. They demonstrate that the API is worth strategy conversion from all offloading to smart offloading.

4.4.4 Threshold Consistency

Once the threshold values are initialized, smart offloading could exploit them to achieve higher efficiency on processing requests. However, we doubt whether threshold values are able to remain the same as networking

or server condition changes.

Therefore, a Perl script is adopted to run on remote desktop in order to cause intensive CPU usage, meanwhile, the Python module for threshold initialization is executed again to create new threshold file.

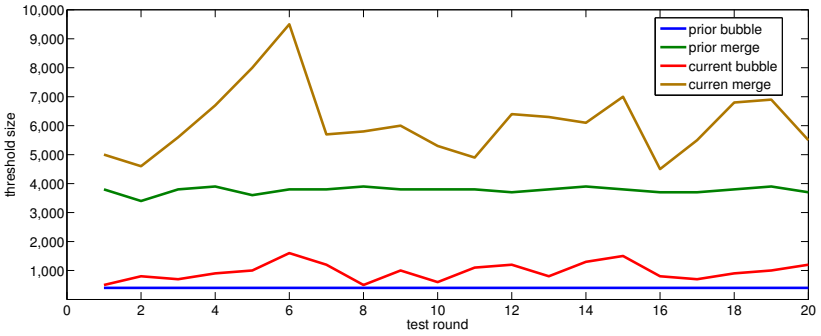


Figure 4.34: Threshold values differences

Figure 4.34 above illustrates that the the threshold consistency is invalid. Since the CPU resource usage on server is in stress, which indicates the CPU capacity is decreased, threshold numbers of both bubble and merge sorting raise to greater values shown as red and brown lines in figure 4.34 that are above blue and green lines separately.

Therefore, timely updating threshold file would improve its accuracy of making offloading decision for smart offloading, as it corresponds to the actual situations.

Chapter 5

Discussion

This project aims to implement a feasible way to improve computing performance for low-cost computers. Thus, an application programming interface has been built by Python to realize computational offloading and it has been tested on the typical resource-poor device, Raspberry Pi.

Due to the fact that the Raspberry Pi has great potential usage in educational ICT environment and it has already been adopted in some schools for basic Computer Science teaching, experiments conducted in this research focus on improving performance in the programming teaching filed.

This chapter will discuss the capability of produced API and interesting findings observed in the interface optimization process. In addition, some introspection and suggestions will be put forward for further research design and implementation.

5.1 Improvement Ability and Scope

As the answer to problem statement in 1.2, the interface has been built up and it has implemented computation offloading for Raspberry Pi, moreover, it has turned out to achieve increased computing performance as well. However, as a matter of fact, this is a exploratory project, and the research process is not always consistent with our initial plans. Thus these following sections will discuss the discrepancy between final version of API and our original expectations.

Affordable Machine vs. Raspberry Pi

The initial goal of this project was deploying computational offloading for improving performances of affordable machines. Therefore, the approach is capable of achieving efficient performance without affecting the low price of those machines.

However, there is no price standards to define 'affordable' and no hardware comparison to explain poor resource. In order to select a typical representative as experiment object within both features at the same time, Raspberry Pi is put into use because of its extremely cheap price as well as extremely poor hardware.

Since the Raspberry Pi has been sold in the market at a low price of several dozens dollars, the weakness of Pi which is originated from limited resource demands to be compared with high-end devices. For the purpose of proving its poor hardware leads to slow performance, benchmark tool 'Sysbench' was adopted to do some tests at the beginning of experiment. The same tool and tests have been done on normal desktop as well to show the performance variation between them and it demonstrates that Raspberry Pi is able to stand for affordable machine well.

Application Performance vs. Computing Performance

As far as we know, the Raspberry Pi has shown its popularity in some individual DIY projects introduced in 2.1.2, the adoption in education domain recently draws more attention due to its profound influence on inspiration and encouragement for empowering children through education.

Thus the research on improving performance demands to be combined with practical status. Programming teaching is one of fundamental requirements in basic CS education, and the original desire of Raspberry Pi foundation is for Python language teaching. So we developed the interface for computing offloading which starts from using Python.

Based on the module SimpleXMLRPCServer in Python, offloading transmission is implemented within the aid of XML-RPC protocol and on the level of procedure call. Therefore for algorithms programming process, the API is easy to expand and realize increased computing performance in future usage, but it is not feasible for every application. Only the application supporting codes, which are compatible with importing Python modules, is able to utilize our API to call remote computational processing.

In addition to that, all our experiment tests were conducted for algorithms programming computing in Python, so computing performance is a better description rather than application performance for this project, even though it is possible to put API into use with other applications in the future.

Increased vs. Optimal

Our goal of this project is always realizing increased performance, but the adoption of our API for computation offloading was unable to keep the

improvement all the time. As the result shown in last chapter, merely offloading computations would lead some processes to perform even worse, especially for some small requests. Thus optimization is required to improve overall performance for API utility.

The optimization process is explained in section 4.4, and it illustrates that whether offloading or not is distinguished by different levels of computing complexity in requests, as it improves overall performance on one hand, and saves networking resource on the other hand.

Thus it is not hard to achieve increased performance under certain condition, but complicated to realize good performance in various situations. The API developed in this project has implemented smart offloading, however, it is still not smart enough and more further investigations are desirable to put into it.

5.2 Exploratory Challenge

Improving performance is required for each system and is the duty for every system administrator. This project aims to improve the performance on Raspberry Pi, but the Pi project has just been implemented in recent two years, thus at the beginning of research, the knowledge for Raspberry Pi was almost blank.

Although the Raspberry Pi is created for school teaching as original desire, it starts to gain the popularity from incredible cost for valuable DIY projects. Thus most of related researches of Raspberry Pi is about getting good performance and functionality within aids of auxiliary equipments for individual usage. However, there is rare research for the combination of improving Pi performance and school teaching as specific task.

However, computing offloading is not a fresh topic and lots of related researches have been turned into practice. Among those interesting investigations, offloading researches for smart phones are most similar to our purpose, as raspberry Pi is also a resource-poor device but expects good performance. Thus computational offloading is selected as the mechanism to implement increased capability.

The mechanism for improving performance of Raspberry Pi had been determined, this topic was still quite open and wide, as performance could be shown in many different aspects. Besides, after Python was chosen to build up the API, the design process was hard to find a start.

Sysbench used in benchmark tests is a good benchmark tool for explaining the performance variation in a easily understandable way. From the benchmark tests data, it demonstrates that the performance of CPU and

memory on Pi is much worse than tested servers.

But when the Pi was taken into CS teaching situation, it performed much better than our expectation. For basic math related calculations, Raspberry Pi does not have much difficulty in dealing with that and the perceived performance was quite good. For instance, when we tried to execute the square root operation with a number of 9 digits in Python application, it only costed less than 1 millisecond that was hard to be perceived by user.

Thus, it needs to look for adequate experiment, which is able to cause intensive CPU usage and perceived latency. After some exploration, array sorting algorithms are found to be a good test instance, not only because they could result in big time latency on Pi, especially bubble sorting, but also because they are very consistent with our purpose to improve performance in education filed.

The idea of smart offloading came from the analysis of offloading performance in sorting algorithms tests. The retrieved data demonstrate that merely offloading is unable to get overall increased computing capability, especially for processing those small requests, which have low computing complexity. It implies that merely offloading has potential weakness, which may result in even worse performance, meanwhile, pollute the networking from providing service for other processes.

Therefore,, the mechanism of smart offloading has been put into use and achieved better performance than merely offloading, as it prejudges the consequence of offloading request first based on threshold file, then determines whether to execute offloading.

However, considering the networking situation changes, threshold values would rise or fall as well. So the final API we created is not adaptive enough. There are several ways in which the API could be improved into smarter versions. For instance, some simple pre-test codes could be executed by offloading each time and the result would determine whether to offload the actual operation. This method constantly adapts to practical condition and better guarantees that each operation would choose the most effective way for execution. But the cost of executing per-test codes each time is also required to be counted into total cost.

Another challenge appeared in experiment is that the API implemented by Python only supports single-threaded processing. Although there was no crush happened to interface through the whole experiment even when receiving frequent requests, it still limits the functionality of server to handle large amount of Pis. Thus implementing API in other languages or optimize API into enabling multi-threads could be investigated in further research.

Moreover, the API adopts Python module 'SimpleXMLRPCServer' to enable offloading, which would cause security issue in transmitting messages, as the module is based on HTTP protocol for transmission without any encryption. So when school decides to deploy an external server for processing offloading computations, the information security demands to be taken into consideration as well.

Since all tests in experiment are designed for being close to education domain, there are lots of other algorithm programming examples that are able to show increased performance rather than only testing bubble and merge sorting. Besides, the other algorithms are quite simple to be added into current API structure.

However, in this project, all attention was on improving computing performance in algorithm programming, which restricts our exploration from investigating performance in many other ways. For instance, what the performance would be if bringing computation offloading to Raspberry Pi when it is running as an image sharing server, and this leads the experiment design to other directions. Meantime, the more important tasks will be image conversion and transmission rather than math computing.

Finally, the performance variation between local and distant server encourages adopting local server to realize offloading, as it saves extra cost on distant transmission. In addition to that, more local desktop nearby could be deployed as offloading agents in practical usage when large number of Pis demand offloading computations. Thus a load balancer needs to be set up in the front end for distributing requests to various agents in the back end. The situation could be described in figure below:

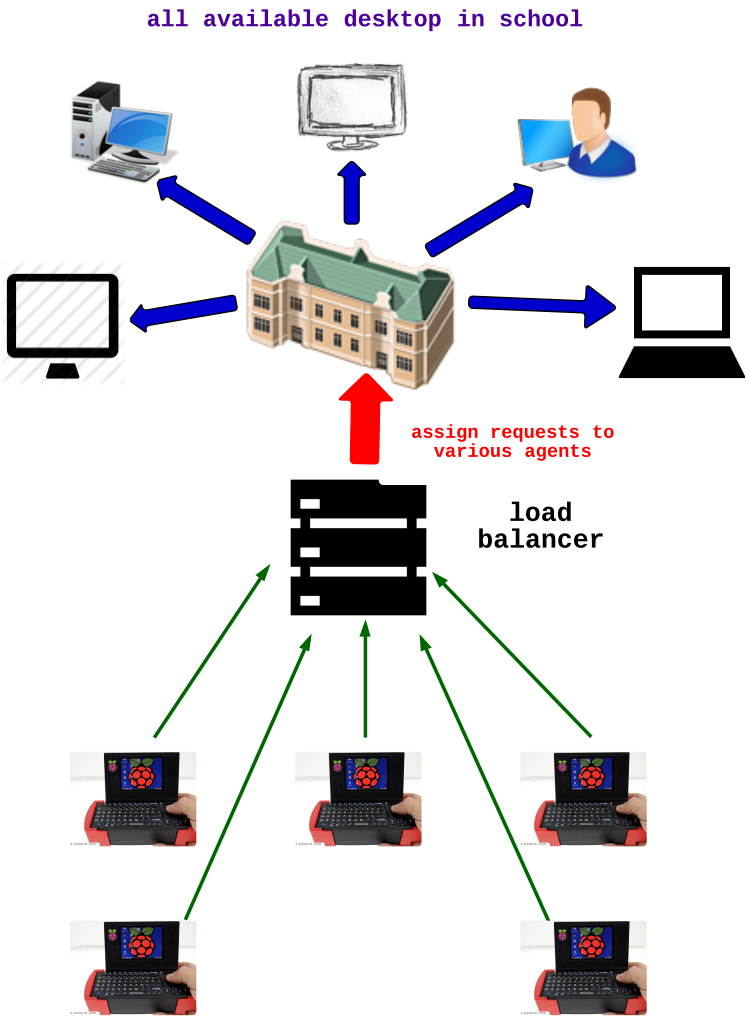


Figure 5.1: Future prospect of deploying all possible agents for offloading computations

5.3 Future API Design

This project designed and implemented an application programming interface for offloading computations from the Raspberry Pi to external server. The API has been proved to achieve increased computing performance for Pi by capability comparison and analysis. Moreover, the mechanism API, with smart offloading adopted, is able to adaptively determine processing computing locally or remotely in order to avoid network pollution by small requests, which is unnecessary to be offloaded. According to these tests

data and analytical results, the mechanism API adopted, smart offloading, is not effective and compatible enough when being used in all applications. The primary reason bringing difficulties to API applicability is the testing domain.

Only algorithm programming process is tested in this project, as we want to make experiment more related to CS teaching. Then Python modules are chosen with the same reason to realize API functionality which turns out to be not powerful enough as it only supports single-threaded processing.

Hence, some suggestions for future research of API implementation are made as following:

- More servers are recommended for testing. When the API is put into use in practical schools, more and more Pis demand to offload computations as the adoption rate increased. Thus servers amount is required to be added as well to deal with large numbers of requests in order to combine actual conditions.

- API demands to be tested in other filed. In this thesis, API is only tested in the situation of Python programming teaching. However, computational offloading would enable increased performance in many other applications. Designing the API to be compatible with other application will increase the adoption rate of Raspberry Pi as well.

- API could be much smarter. It demands to look for other novel ways to implement the API to be more effective, even adaptive to determine each operation whether offloading or not in real time, which enables better perceived performance.

- Explore other APIs. Implementing API by other languages may enable multi-threaded computing processes, or developing Python API to achieve the same goal is another choice.

Chapter 6

Conclusion

This thesis aims to achieve increased application performance on the Raspberry Pi affordable computing device by offloading native computations to external server through an API. Major tasks have been accomplished in this project as the description below:

- An API has been developed by Python for successfully realizing computational offloading.
- Empty request test has been implemented and analyzed for describing transmission cost on offloading.
- Message size test has been developed and implemented for analyzing the impact of different request sizes on transmission cost.
- Increased level of computing complexity and sorting algorithm tests have been accomplished and analyzed for putting API into effect on practical algorithm programming teaching situation. Then the retrieved data has been discussed for performance comparison.
- API has been optimized based on the discussion of generated data, by deploying smart offloading mechanism in order to achieve better efficiency.
- Recommendations and suggestions about API design and implementation for future project have been proposed.

The implemented API enables increased application performance of Raspberry Pi and adaptive mechanism for efficient usage of networking resource. But the emphasis on programming teaching in experiment narrow down the applicability of API. Thus, further research has high potential to explore more efficient interface for offloading computations.

Bibliography

[1] *About us*. http://www.raspberrypi.org/about. 2012.

[2] *APC was not built like ordinary PCs*. http://apc.io/products/. Jan. 2013.

[3] Robert J Barro. 'Education and economic growth'. In: *Harvard University* (2000).

[4] David M Beazley. *Python: essential reference*. Sams, 2006.

[5] Michael Clemens. *The One Button Audiobook Player*. http://blogs.fsfe.org/clemens/2012/10/30/the-one-button-audiobook-player/. Oct. 2012.

[6] *DATWIND Ltd*. http://www.ubislate.com/index.html. Jan. 2013.

[7] LAPTOP Contributor Davey Alba. *The 8 Cheapest Computers in the World*. http://blog.laptopmag.com/the-8-cheapest-computers-in-the-world?slide=1. May 2012.

[8] *Dropbox*. https://www.dropbox.com/. 2013.

[9] L.L. Ferreira, G. Silva and L.M. Pinho. 'Service offloading in adaptive real-time systems'. In: *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*. IEEE. 2011, pp. 1–6.

[10] Richard Hogg. 'ICT education: the challenge for the future'. In: *Proceedings of the 3.1 and 3.3 working groups conference on International federation for information processing: ICT and the teacher of the future - Volume 23*. CRPIT '03. Melbourne, Australia: Australian Computer Society, Inc., 2003, pp. 57–59. ISBN: 1-920682-02-3. URL: http://dl.acm.org/citation.cfm?id=857097.857114.

[11] Shigeru Imai and Carlos A. Varela. 'Light-weight adaptive task offloading from smartphones to nearby computational resources'. In: *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*. RACS '11. Miami, Florida: ACM, 2011, pp. 146–152. ISBN: 978-1-4503-1087-1. DOI: 10.1145/2103380.2103411. URL: http://doi.acm.org/10.1145/2103380.2103411.

[12] Yaacov J Katz. 'Attitudes affecting college students' preferences for distance learning'. In: *Journal of Computer Assisted Learning* 18.1 (2002), pp. 2–9.

[13] R Kozma. 'ICT, education reform, and economic growth'. In: *Retrieved October* 19 (2005), p. 2009.

[14] O.R.Senthil Kumaran. *Algorithms in Python*. Sept. 2009.

[15] K. Kumar and Y.H. Lu. 'Cloud computing for mobile users: Can offloading computation save energy?' In: *Computer* 43.4 (2010), pp. 51–56.

[16] *Linux Programmer's Manual*. http://man7.org/linux/man-pages/man5/proc.5.html. Apr. 2013.

[17] liz. *15,000 Raspberry Pis for UK schools - thanks Google!* http://www.raspberrypi.org/archives/3158. Jan. 2013.

[18] Z. Li, C. Wang and R. Xu. 'Computation offloading to save energy on handheld devices: a partition scheme'. In: *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM. 2001, pp. 238–246.

[19] *Mele A1000 is a $70 hackable, Linux-friendly ARM-based PC*. http://liliputing.com/2012/03/mele-a1000-is-a-70-hackable-linux-friendly-arm-based-pc.html. Mar. 2012.

[20] *Miniand Ltd.* https://www.miniand.com/. Jan. 2013.

[21] Jeremy Morgan. *Tutorial: How to Set Up a Raspberry Pi Web Server*. http://www.jeremymorgan.com/tutorials/raspberry-pi/how-to-raspberry-pi-web-server/. Nov. 2012.

[22] Jeremy Morgan. *Tutorial: How to Set Up a Raspberry Pi Web Server*. http://www.jeremymorgan.com/blog/programming/raspberry-pi-web-server-comparison/. Dec. 2012.

[23] *one laptop per child*. http://one.laptop.org/.

[24] *ownCloud*. http://owncloud.org/. 2013.

[25] Miss Philbin. *Deploying Raspberry Pis in a Classroom - Problems & Solutions*. http://misscphilbin.blogspot.co.uk/2013/04/deploying-raspberry-pis-in-classroom.html?m=1. Apr. 2013.

[26] George Psacharopoulos and Harry Patrinos. 'Returns to investment in education: A further update'. In: *World Bank policy research working paper* 2881 (2002).

[27] *Raspberry Pi Owncloud (dropbox clone)*. http://www.instructables.com/id/Raspberry-Pi-Owncloud-dropbox-clone/?ALLSTEPS. 2012.

[28] *Raspbian*. http://www.raspbian.org/. Jan. 2013.

[29] Mahadev Satyanarayanan et al. 'The Case for VM-Based Cloudlets in Mobile Computing'. In: *IEEE Pervasive Computing* 8.4 (Oct. 2009), pp. 14–23. ISSN: 1536-1268. DOI: 10.1109/MPRV.2009.82. URL: http://dx.doi.org/10.1109/MPRV.2009.82.

[30] Barbara Sianesi and John Van Reenen. 'The returns to education: a review of the empirical macro-economic literature'. In: (2002).

[31] C. Wang and Z. Li. 'Parametric analysis for adaptive computation offloading'. In: *ACM SIGPLAN Notices* 39.6 (2004), pp. 119–130.

[32] C. Xian, Y.H. Lu and Z. Li. 'Adaptive computation offloading for energy conservation on battery-powered systems'. In: *Parallel and Distributed Systems, 2007 International Conference on*. Vol. 2. IEEE. 2007, pp. 1–8.

Appendix

Scripts for testing API utility

python_server.py

```
1  #!/usr/bin/python -tt
2
3  import SimpleXMLRPCServer
4  import math
5
6  def add_all_number(n):
7      count=0.0
8      for i in range(1,n):
9          number=1.0/i
10         count+=number
11     a="%.3f"%(count)
12     return a
13
14 def return_array_quantity(data):
15     a=len(data)
16     sort_array=[]
17     sort_array=sorted(data)
18     return a
19
20 def return_array(data):
21     a=len(data)
22     sort_array=[]
23     sort_array=sorted(data)
24     return sort_array
25
26 def small_request():
27     return
28
29 def bubblesort(A):
30     swapped = True
31     while swapped:
32         swapped = False
33         for i in range(len(A)-1):
34             if A[i] > A[i+1]:
```

```
35                  A[i], A[i+1] = A[i+1], A[i]
36                  swapped = True
37      return A
38
39  def merge(left, right):
40      result = list()
41      while (len(left) > 0) and (len(right) > 0):
42          if left[0] <= right[0]:
43              result.append(left[0])
44              left = left[1:]
45          else:
46              result.append(right[0])
47              right = right[1:]
48
49      if left: result.extend(left)
50      else:    result.extend(right)
51      return result
52
53  def mergesort(m):
54      left = list()
55      right = list()
56      result = list()
57      if len(m) <= 1: return m
58
59      middle = int(math.ceil(len(m)/2.0))
60      for x in range(0,middle):
61          left.append(m[x])
62      for x in range(middle,len(m)):
63          right.append(m[x])
64
65      left = mergesort(left)
66      right = mergesort(right)
67      left_last_item = left[len(left)-1]
68      right_first_item = right[0]
69      if left_last_item > right_first_item:
70          result = merge(left, right)
71      else:
72          left.extend(right)
73          result = left
74      return result
75
76  s = SimpleXMLRPCServer.SimpleXMLRPCServer
77  (("mime.iu.hio.no",8080),allow_none=True)
78  s.register_function(add_all_number)
79  s.register_function(return_array_quantity)
80  s.register_function(return_array)
81  s.register_function(small_request)
82  s.register_function(bubblesort)
```

```
83 s.register_function(mergesort)
84 s.register_instance(math)
85 s.register_introspection_functions()
86 s.register_multicall_functions()
87 s.serve_forever()
```

python_test.py

```
1  #!/usr/bin/python -tt
2
3  import xmlrpclib
4  import sys
5  import time
6  import datetime
7  import random
8  import re
9
10 def write_data(data):
11     now=datetime.datetime.utcnow()
12     (month,day,hour,minute)=(now.month,now.day,
13                              now.hour,now.minute)
14     filename="data/code_"+str(month)+"_"
15     +str(day)+"_"+str(hour)+"_"+str(minute)
16     f=open(filename,'w')
17
18     # ------------------------------------------
19     for (i,j) in sorted(data.items()):
20         f.write(str(i)+'\t'+str(j)+'\n')
21     # ------------------------------------------
22     n=1
23     for i in data:
24         if n%3==0: f.write(str(i)+'\n')
25         else: f.write(str(i)+'\t')
26         n=n+1
27     # ------------------------------------------
28     for t in data:
29         match=re.search(r'[^.]+.[0]*(\d+)',str(t))
30         m=match.group(1)
31         f.write(m+'\n')
32     # ------------------------------------------
33     f.close()
34
35 def main():
36     s=xmlrpclib.ServerProxy("http://mime.iu.hio.no:8080")
37     # n=[10,100,1000,10000,100000,200000,500000,
38     #    800000,1000000]
39     # n=[10,100,1000,2000,3000,4000,5000,6000,7000,
```

```
40      #      8000,9000,10000]
41      # ------------------------------------------------------
42      # increasing complexity test
43      data={}
44      for i in n:
45          t1=time.time()
46          for j in range(20):
47              count=s.add_all_number(i)
48          t2=time.time()
49          t='%.5f'%(t2-t1)
50          data[i]=t
51      write_data(data)
52      # ------------------------------------------------------
53      # request size test
54      total_time=[]
55      for i in n:
56          data=[]
57          print i
58          total_time.append(i)
59          for j in range(i):
60              x=random.uniform(1,1000)
61              y='%.3f'%(x)
62              data.append(y)
63          time.sleep(1)
64          m1=time.time()
65          for a in range(1):
66              count=s.return_array_quantity(data)
67          m2=time.time()
68          t1='%.5f'%(m2-m1)
69          total_time.append(t1)
70
71          n1=time.time()
72          for b in range(1):
73              value=[]
74              value=s.return_array(data)
75          n2=time.time()
76          t2='%.5f'%(n2-n1)
77          total_time.append(t2)
78      write_data(total_time)
79      # ------------------------------------------------------
80      # test small_request
81      data=[]
82      for i in range(1000):
83          a=datetime.datetime.now()
84          s.small_request()
85          b=datetime.datetime.now()
86          c=b-a
87          data.append(c)
```

```
88      write_data(data)
89  if __name__ == '__main__':
90          main()
```

sort_algorithm.py

```
1   #!/usr/bin/python -tt
2
3   import random
4   import math
5   import datetime
6   import time
7   import xmlrpclib
8   import re
9
10  def write_data(T):
11      now=datetime.datetime.utcnow()
12      (month,day,hour,minute)=(now.month,now.day,
13                               now.hour,now.minute)
14      filename="data/test_algorithm/baseline_"+
15      str(month)+"_"+str(day)+"_"+str(hour)+"_"+str(minute)
16      f=open(filename,'w')
17      n=1
18      for i in T:
19          if n%3==0: f.write(str(i)+'\n')
20          else: f.write(str(i)+'\t')
21          n=n+1
22      f.close()
23
24  def bubblesort(A):
25      swapped = True
26      while swapped:
27          swapped = False
28          for i in range(len(A)-1):
29              if A[i] > A[i+1]:
30                  A[i], A[i+1] = A[i+1], A[i]
31                  swapped = True
32      return A
33
34  def merge(left, right):
35      result = list()
36      while (len(left) > 0) and (len(right) > 0):
37          if left[0] <= right[0]:
38              result.append(left[0])
39              left = left[1:]
40          else:
41              result.append(right[0])
```

```
42              right = right[1:]
43
44      if left: result.extend(left)
45      else:    result.extend(right)
46      return result
47
48  def mergesort(m):
49      left = list()
50      right = list()
51      result = list()
52      if len(m) <= 1: return m
53
54      middle = int(math.ceil(len(m)/2.0))
55      for x in range(0,middle):
56          left.append(m[x])
57      for x in range(middle,len(m)):
58          right.append(m[x])
59
60      left = mergesort(left)
61      right = mergesort(right)
62      left_last_item = left[len(left)-1]
63      right_first_item = right[0]
64      if left_last_item > right_first_item:
65          result = merge(left, right)
66      else:
67          left.extend(right)
68          result = left
69      return result
70
71  def main():
72      n=[10,50,100,300,500,700,900,1000,2000,3000,5000]
73      s=xmlrpclib.ServerProxy("http://mime.iu.hio.no:8080")
74      time_data=[]
75      for i in n:
76          time_data.append(i)
77          print i
78          array=[]
79          # use bubble sort
80          for j in range(i):
81              a=random.uniform(1,1000)
82              b='%.3f'%(a)
83              array.append(b)
84          x1=time.time()
85          for r in range(20):
86              B=[]
87              B=s.bubblesort(array)
88          y1=time.time()
89          z1='%.6f'%(y1-x1)
```

```
90          print "bubble:\t"+str(z1)
91          time_data.append(z1)
92          # use merge sort
93          array=[]
94          for j in range(i):
95              a=random.uniform(1,1000)
96              b='%.3f'%(a)
97              array.append(a)
98          x2=time.time()
99          for r in range(20):
100             C=[]
101             C=s.mergesort(array)
102         y2=time.time()
103         z2='%.6f'%(y2-x2)
104         print "merge:\t"+str(z2)
105         time_data.append(z2)
106     write_data(time_data)
107
108 if __name__ == '__main__':
109     main()
```

experiment.pl

```
1  #!/usr/bin/perl
2
3  use Getopt::Std;
4  use strict;
5  use feature "say";
6
7  getopts('hbpsc',\ my %opts) || usage();
8
9  # validate all arguments
10 &arg_valid(%opts);
11
12 # running baseline test
13 if(exists $opts{"b"}){
14     if(!-d "data/"){ say "There is no dir '~/data/'
15 to store data, please make it first!!"; exit;}
16     if(!-f "sort_algorithm.py")
17      { say "There is no baseline script, please copy
18 it to current dir!!"; exit;}
19     if(exists $opts{"c"}){
20   say "The baseline test on client begins:";
21   system("./monitor.pl -c sort_algorithm.py -i 1 &");
22         if(!system("./sort_algorithm.py"))
23   { say "Baseline test on client is done!"; exit;}
24     }
```

```
25      else{
26      say "The baseline test on server begins:";
27              if(!system("./baseline.py server"))
28      { say "Baseline test on server is done!"; exit;}
29      }
30  }
31
32  # running python code test on client
33  if(exists $opts{"p"}){
34      if(!-d "data/")
35      { say "There is no dir '~/data/' to store data,
36  please make it first!!"; exit;}
37      if(!-f "sort_algorithm.py")
38      { say "There is no code testing script,
39  please copy it to current dir!!"; exit}
40      say "The python code test on client begins:";
41      system("./monitor.pl -c sort_algorithm.py -i 1 &");
42      if(!system("./sort_algorithm.py"))
43      {say "Python code test on client is done!"; exit;}
44  }
45
46  sub arg_valid(){
47      my %opt=@_;
48      if(exists $opt{"h"}){ usage(); }
49      if(exists $opt{"b"} && exists $opt{"p"}){
50    say "You could only run one kind of tests each time!";
51    say "Baseline test or API utility test";
52    usage();
53      }
54      if((exists $opt{"s"} && !exists $opt{"b"})
55  || (exists $opt{"c"} && !exists $opt{"b"})){
56    say "Wrong Input!!!";
57    usage();
58      }
59      if(exists $opt{"b"} && !exists $opt{"s"}
60  && !exists $opt{"c"}){
61    say "Wrong Input!!!";
62    usage();
63      }
64  }
65
66  sub usage{
67      say "-------------------------------------------------------";
68      say "There are 5  arguments for running this script";
69      say "-h script usage information";
70      say "-b flag to execute the baseline test";
71      say "-s flag to do the server baseline test,
72  use with -b";
```

```
73      say "-c flag to do the client baseline test,
74  use with -b";
75      say "-p flag to execute the API utility test";
76      say "-------------------------------------------------------";
77      exit;
78  }
```

process_data.pl

```
1   #!/usr/bin/perl
2
3   use Getopt::Std;
4   use strict;
5   use feature "say";
6
7   getopts('c:s:',\ my %opts) || usage();
8
9   my $file1=$opts{"c"};
10  my $file2=$opts{"s"};
11
12  my @data1;
13  my @data2;
14
15  open(ONE,$file1);
16  while(<ONE>){
17      my $data; my $line=$_;
18      $line =~ /[^\t]+\t([0-9.]+)$/;
19      $data=$1;
20      say "$data";
21      push(@data1,$data);
22  }
23  close(ONE);
24
25  open(TWO,$file2);
26  while(<TWO>){
27      my $data; my $line=$_;
28      $line=~/^[0-9]+\t([.0-9]+)$/;
29      $data=$1;
30      say "$data";
31      push(@data2,$data);
32      next;
33  }
34  close(TWO);
35
36  my @trip; my $i;
37  for($i=0;$i<$#data1;$i++){
38      my $value=$data1[$i]-$data2[$i];
```

```
39      push(@trip,$value);
40  }
41
42  my @n=(10,100,1000,10000,100000,200000,500000,
43  800000,1000000);
44  my $file="$file1\_$file2";
45  open(WRITE,">",$file);
46  for($i=0;$i<$#trip;$i++){ say WRITE "$n[$i]\t$trip[$i]"; }
47  close(WRITE);
48
49  sub usage{
50      say "You␣two␣arguments␣to␣run␣this␣script:";
51      say "-c␣the␣name␣of␣client␣data␣file";
52      say "-s␣the␣name␣of␣server␣data␣file";
53      exit;
54  }
```

monitor.pl

```
1   #!/usr/bin/perl
2
3   use strict;
4   use feature "say";
5   use Getopt::Std;
6
7   getopts('c:i:',\ my %opts) || usage();
8   my $command = $opts{c};
9   my $interval= $opts{i};
10  my $running=1;
11  my @data;
12
13  my $pid=getPid($command);
14  if ($pid == ''){say "The␣input␣command␣is␣not␣running";
15  exit;}
16  while($running==1){
17      if(-d "/proc/$pid/"){
18    my $time=time();
19    # say "$time: $pid is alive!";
20    # collect the data
21    open(STATUS,"/proc/$pid/status");
22    while(<STATUS>){
23          next until /^VmSize:[^\d]+([0-9]+)/;
24          my $value=$1;
25          push(@data,$value);
26          <STATUS>;<STATUS>;<STATUS>;<STATUS>;
27          my $line=$_;
28          $line =~ /VmRSS:[^\d]+([0-9]+)/;
```

```
29          my $value=$1;
30          push(@data,$value);
31          last;
32      }
33    close(STATUS);
34
35    open(STAT,"/proc/$pid/stat");
36    my $line=<STAT>;
37    my @stat_value=split(/\s/,$line);
38    push(@data,$stat_value[9]);
39    push(@data,$stat_value[11]);
40    close(STAT);
41
42    open(SCHED,"/proc/$pid/sched");
43    while(<SCHED>){
44          $line=$_; my $value;
45          if($line =~ /^se.sum_exec_runtime[^\d]+([.0-9]+)/)
46          {$value=$1;push(@data,$value);next;}
47          if($line =~ /^se.statistics.wait_sum[^\d]+([.0-9]+)/)
48          {$value=$1;push(@data,$value);next;}
49          if($line =~ /^se.statistics.iowait_sum[^\d]+([.0-9]+)/)
50          {$value=$1;push(@data,$value);next;}
51          if($line =~ /^nr_involuntary_switches[^\d]+([0-9]+)/)
52          {$value=$1;push(@data,$value);last;}
53    }
54            close(SCHED);
55    push(@data,"\n");
56      } else{ $running = 0; }
57      sleep($interval);
58      # last;
59  }
60
61  my $sec; my $min; my $hour;my $mday; my $mon;
62  ($sec,$min,$hour,$mday,$mon) = localtime(time);
63  open(WRITE,'>',"data/monitor/
64  $mon\_$mday\_$hour\_$min\_$sec.txt");
65  foreach my $value(@data){printf WRITE "\t$value";}
66            # if($value == "\n"){printf WRITE "$value";}
67            # else{printf WRITE "$value\t";}
68
69  sub getPid{
70      my $process=$_[0];
71      open(PS,"ps␣aux␣|");
72      my $line;
73      while($line = <PS>){
74    if($line =~ /^\S+\s+(\d+)\s.*.\/$process/){
75          my $pid=$1;
76          close(PS);
```

```
77          return $1;
78      }
79          }
80  }
81
82  sub usage{
83          say "Wrong argument input!!";
84          exit;
85  }
```

Scripts of smart offloading

Jun.py

```
1   #!/usr/bin/python -tt
2
3   import sys
4   import random
5   import time
6   import xmlrpclib
7   import re
8   import Algorithms
9   import os
10
11  def write_data(server,name,value):
12      filename="data/threshold_"+server
13      f=open(filename,'a')
14      if name=='bubble':
15          f.write('bubblesort\t'+str(value)+'\n')
16      if name=='merge':
17          f.write('mergesort\t'+str(value)+'\n')
18      f.close()
19
20  def return_threshold():
21      f=open('data/threshold_mime.iu.hio.no','rU')
22      lines=f.readlines()
23      f.close()
24      i=0
25      for line in lines:
26          match=re.search(r'[a-z]+\t(\d+)',line)
27          if i==0: bubble=match.group(1)
28          else: merge=match.group(1)
29          i=i+1
30      return int(bubble),int(merge)
31
32  def bubblesort(A):
```

```
33      size=len(A)
34      (thre_bubble,thre_merge)=return_threshold()
35      if size<=thre_bubble:
36          return Algorithms.bubblesort(A)
37      else:
38          s=xmlrpclib.ServerProxy
39          ("http://mime.iu.hio.no:8080")
40          return s.bubblesort(A)
41
42  def mergesort(A):
43      size=len(A)
44      (thre_bubble,thre_merge)=return_threshold()
45      if size<=thre_merge: return Algorithms.mergesort(A)
46      else:
47          s=xmlrpclib.ServerProxy
48          ("http://mime.iu.hio.no:8080")
49          return s.mergesort(A)
50
51  def init(server,interval):
52      s=xmlrpclib.ServerProxy("http://"+server+":8080")
53      bubble=bool(1)
54      merge=bool(1)
55      start=0
56      while bubble or merge:
57          start=start+int(interval)
58
59          # find threshold of bubble
60          if bubble:
61              array=[]
62              for j in range(start):
63                  a=random.uniform(1,1000)
64                  b='%.3f'%(a)
65                  array.append(b)
66              B=[]
67              x1=time.time()
68              B=Algorithms.bubblesort(array)
69              y1=time.time()
70              z1=20*(y1-x1)
71              # offloading processing
72              x2=time.time()
73              for i in range(20):
74                  B=[]
75                  B=s.bubblesort(array)
76              y2=time.time()
77              z2=y2-x2
78              if z1<=z2: bubble=bool(1)
79              else:
80                  print str(start)
```

```
81                  print 'Pi:\t'+str(z1)
82                  print 'offload:\t'+str(z2)
83                  bubble=bool(0)
84                  thres_bubble=start
85                  write_data(server,'bubble',thres_bubble)
86
87          # find threshold of mergesort
88          if merge:
89              m2=0.0
90              array=[]
91              for j in range(start):
92                  a=random.uniform(1,1000)
93                  b='%.3f'%(a)
94                  array.append(b)
95              C=[]
96              t1=time.time()
97              C=Algorithms.mergesort(array)
98              t2=time.time()
99              m1=20*(t2-t1)
100             # offloading processing
101             for i in range(20):
102                 C=[]
103                 t1=time.time()
104                 C=s.mergesort(array)
105                 t2=time.time()
106                 m2=(t2-t1)+m2
107             if m1<=m2: merge=bool(1)
108             else:
109                 print str(start)
110                 print 'Pi:\t'+str(m1)
111                 print 'offload:\t'+str(m2)
112                 merge=bool(0)
113                 thres_merge=start
114                 write_data(server,'merge',thres_merge)
115     print str(thres_bubble)+'\t'+str(thres_merge)
116
117 def main():
118     server=sys.argv[1]
119     interval=sys.argv[2]
120     init(server,interval)
121
122 if __name__ == '__main__':
123     main()
```

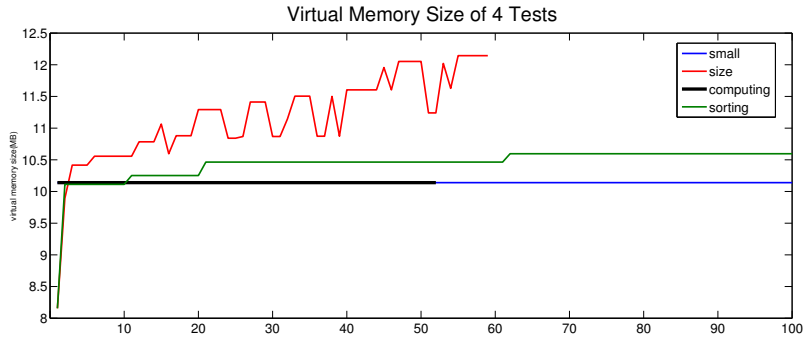
smart_test.py

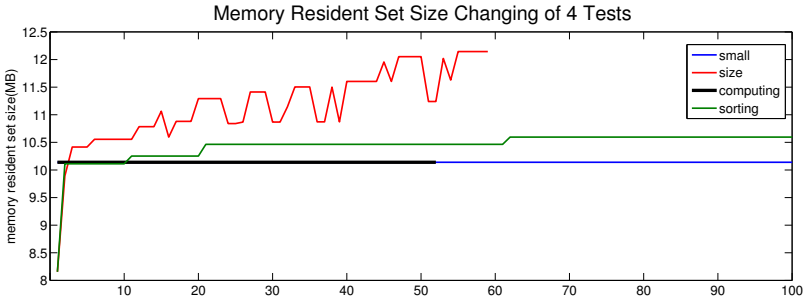
```
1 #!/usr/bin/python -tt
```

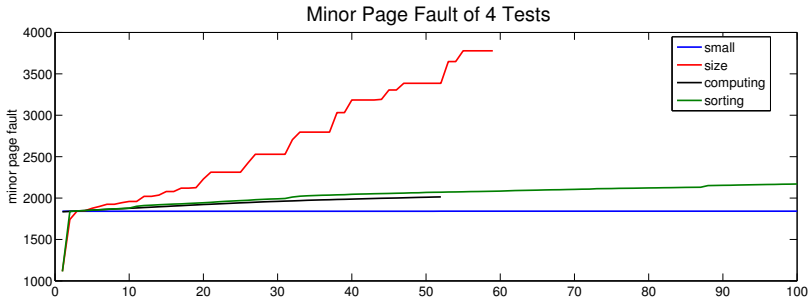
```
2
3  import Algorithms
4  import Jun
5  import time
6  import random
7  import xmlrpclib
8  import sys
9
10 def main():
11     s=xmlrpclib.ServerProxy("http://mime.iu.hio.no:8080")
12     pi=0.0
13     normal=0.0
14     smart=0.0
15     # bubble sort testing
16     print '-------------------------------------------------'
17     print "Smart Offloading test for bubble sorting:"
18     for i in range(20):
19         size=random.randint(0,200)
20         # print 'Array size:'+str(size)
21         array=[]
22         for j in range(size):
23             a=random.uniform(1,1000)
24             b='%.3f'%(a)
25             array.append(a)
26         C=[]
27         x2=time.time()
28         C=Jun.bubblesort(array)
29         y2=time.time()
30         z2='%.6f'%(y2-x2)
31         smart=smart+float(z2)
32         B=[]
33         x1=time.time()
34         B=s.bubblesort(array)
35         y1=time.time()
36         z1='%.6f'%(y1-x1)
37         normal=normal+float(z1)
38         array=[]
39         for j in range(size):
40             a=random.uniform(1,1000)
41             b='%.3f'%(a)
42             array.append(a)
43         D=[]
44         x3=time.time()
45         D=Algorithms.bubblesort(array)
46         y3=time.time()
47         z3='%.6f'%(y3-x3)
48         pi=pi+float(z3)
49     print 'Without offloading\tAll offloading\t
```

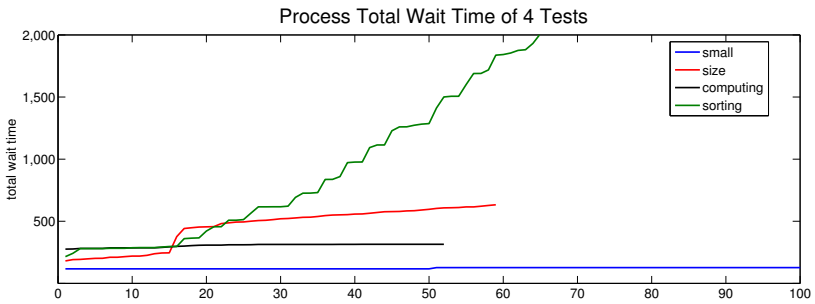
```
50 ␣␣␣␣␣␣␣␣␣␣␣Smart␣offloading'
51     print str(pi)+'\t'+str(normal)+'\t'+str(smart)
52
53     print '-------------------------------------------------------'
54     print "Smart␣Offloading␣test␣for␣merge␣sorting:"
55     pi=0.0
56     normal=0.0
57     smart=0.0
58     for i in range(20):
59         size=random.randint(0,2000)
60         # print 'Array size:'+str(size)
61         array=[]
62         for j in range(size):
63             a=random.uniform(1,1000)
64             b='%.3f'%(a)
65             array.append(a)
66         C=[]
67         x2=time.time()
68         C=Jun.mergesort(array)
69         y2=time.time()
70         z2='%.6f'%(y2-x2)
71         smart=smart+float(z2)
72         B=[]
73         x1=time.time()
74         B=s.mergesort(array)
75         y1=time.time()
76         z1='%.6f'%(y1-x1)
77         normal=normal+float(z1)
78         D=[]
79         x3=time.time()
80         D=Algorithms.mergesort(array)
81         y3=time.time()
82         z3='%.6f'%(y3-x3)
83         pi=pi+float(z3)
84
85     print 'Without␣offloading\tAll␣offloading\t
86 ␣␣␣␣␣␣␣␣␣␣␣Smart␣offloading'
87     print str(pi)+'\t'+str(normal)+'\t'+str(smart)
88
89 if __name__ == '__main__':
90     main()
```

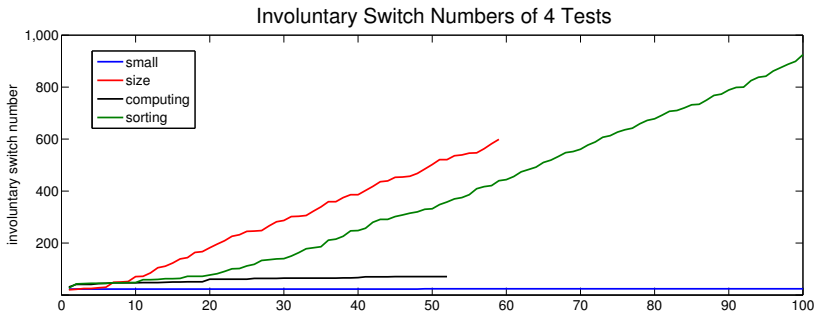
Monitoring results from offloading to remote desktop










Involuntary Switch Numbers of 4 Tests
involuntary switch number
1,000
800
600
400
200
0
10
20
30
40
50
60
70
80
90
100
small
size
computing
sorting