Energy Performance Analysis of Software Applications on Servers

by

Jasmeet Singh

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2014

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Jasmeet Singh

Abstract

The power cost of running a data center is a significant portion of its total annual operating budget. Although the hardware subsystems, namely, processors, memory, disk, and network interfaces of a server actually consume power, it is the software activities that drive the operations of the hardware subsystems leading to varying dynamic power cost. With the aim of reducing power bills of data centers, "Green Computing" has emerged with the primary goal of making software more energy efficient without compromising the performance. Developers play an important role in controlling the energy cost of data center software while writing code. Bearing green principles in mind during design and coding stages of the software life-cycle can have a great impact on the energy efficiency of the final software product. There are a number of ways to optimize application programs at their design stages but it is difficult for the developers to analyse their applications in terms of power cost on the real servers. Reading big data, moving large amount of data from one server to another, compressing data to gain storage space, and decompressing it back are some key operations that are performed extensively on large scale servers in data centers.

In the first part of this thesis, we present the design of an automated test bench to measure the power cost of an application running on a server. We show how our test bench can be used by software developers to measure and improve the energy cost of two Java file access methods. Another benefit of our test bench has been demonstrated by comparing the energy footprint measurements of compression and decompression features provided by two popular Linux packages: *7z* and *rar*. This information will be helpful in choosing a Green Software among others to perform a desired function.

In the second part, we show how software developers can contribute to energy efficiency of servers by choosing energy efficient APIs (Application Programming Interface) with the optimal choice of parameters while implementing file reading, file copy, file compression and file decompression operations in Java. We performed extensive measurements of energy cost of those operations on a Dell Power Edge 2950 machine running Linux and Windows servers. Measurement results show that energy costs of various APIs for those operations are sensitive to the *buffer size* selection. The choice of a particular Java API for file reading with different buffer sizes has significant impact on the energy cost, giving an opportunity to save up to 76%. To save energy while copying files, it is important to use APIs with tunable buffer sizes, rather than APIs using fixed size buffers. In addition, there is a trade off between compression ratio and energy cost: because of higher compression ratio, *xz* compression API consumes more energy than *zip* and *gzip* compression APIs.

The third part of the thesis presents a design of a framework in which one developer generates energy cost models for the common design options. Afterwords, other developers can make use of those models to find the energy costs for the same design options instead of direct measurements.

Overall, this thesis makes a contribution to reduce the perception gap between high level programs and the concept of energy efficiency.

Acknowledgements

I would like to thank all the people who made this thesis possible.

Dedication

to my beloved parents S. Gurmukh Singh and Harjit Kaur.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Electrical energy is a key resource consumed by all computing platforms [33][35]. The increasing demand of computing resources of today' s data centers lead to important issues in energy management as the power cost to run a data center is a significant portion of its total annual operating cost. The total electricity used by data centers in 2010 was about 1.3% of all electricity consumed in the world, and about 2% of all electricity used in the United States [27]. The energy cost is expected to be even higher in the future. Xu et al. [51] claimed that electricity consumed by the servers and cooling systems in a typical data center accounts for about 20% of the total operating cost. In addition to the high cost of operations and maintenance of data centers, such huge power consumption is also detrimental to the environment. Modern data centers hosting big applications like Google Search, Facebook, Google Mail (Gmail), Amazon Web Services, Twitter, etc. consume a major portion of world's electrical energy because these applications are centred on processing large data. In addition, those cloud serving systems have large-scale data storage requirements and are being used in social networking, music downloads, video-on-demand, telephony, business intelligence and web 2.0 services. The results of one report [32] are shown in the Figure 1.1 which tells that amongst all the leading cloud services, Facebook has the largest data center power usage.

To reduce the cost of data centers, much progress has been made in improving the energy efficiency of hardware and operating systems [22][40]. However, in the last few years researchers started focussing to the energy impact of software because software activities have a direct influence on the energy consumption of hardware underneath [9, 36]. It is the software activities that drive the operations of the hardware subsystems namely, disk, CPU, and memory; leading to varying dynamic power cost. As a result, the term "Green Computing" has emerged with the primary goal of making software more energy efficient

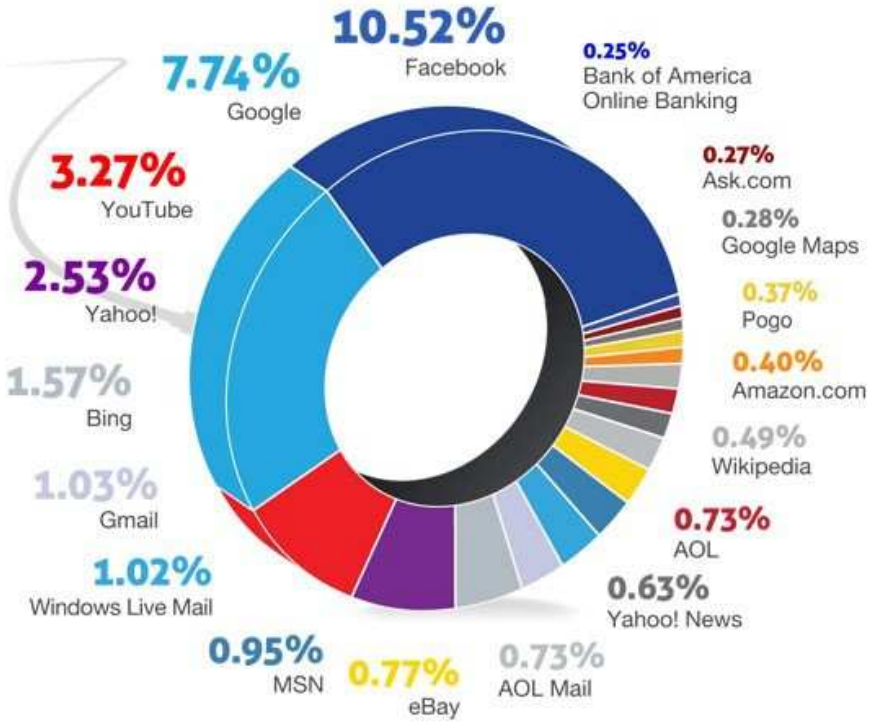

Figure 1.1: Data Centers Power Usage (Source: *http://www.engadget.com/2011/04/26/visualized-ring-around-the-world-of-data-center-power-usage)*

without compromising the performance. Green Computing involves a mixture of new approaches for power and cooling with energy-efficient hardware, virtualization, software, and power and workload management [21]. Adel et al. [14] analyzed all the activities of life-cycle of software development and recommended that green principles should be followed in all the activities of software development life-cycle to reduce the impact of software on energy cost. Therefore, Energy Efficiency has become a key factor in software development.

1.1 Problem Statement

The design of a software application has a significant impact on the power consumption [41][3]. Various techniques have been suggested to reduce the power consumption of mobile devices in general [34] and software systems in particular [41][7]. Considering the fact that

power bill accounts for a significant portion of the cost to run a data center, it is useful to analyse and minimize the energy cost of applications running on large systems, namely, servers. Although there are a number of ways to optimize the application at its design stage, developers generally do not consider the energy cost of their software while making important design decisions. They find it difficult to measure the energy cost incurred by their workload and know how it behaves on real servers inside data centers. In addition to this, the measurement process takes a lot of human effort and time. The main problem that we are trying to tackle through this thesis is:

How developers can accurately evaluate the energy performance of applications for the various design choices on a server during the coding stage of software lifecycle?

1.2 Solution Strategy and Contributions

We develop an automated test bench to measure the power cost of an application running on a server. The test bench comprises a high precision power meter, a monitoring computer to acquire power readings from the power meter, and a newly developed control software: **PAST** to synchronize the metering system with the load running on the server. Our test bench can be used by developers to measure the energy cost of their applications for the various design choices and can come up with an energy efficient design. Another benefit of test bench is that it can be used to measure and compare the energy footprints of the same functionality provided by two different applications. This is useful in choosing the energy efficient software applications among others to use in data centers.

Following are the main contributions of our work:

1. showing the importance of our test bench to developers in measuring the energy cost of their applications for the various design choices and can come up with an energy efficient design.
2. studying the impact of developer choices of Java APIs and buffer size in implementing file reading, file copy, file compression and decompression operations on the energy cost of servers.
3. presenting a design of framework which avoids repeated energy measurements for the common design by making use of mathematical models of the energy costs.

1.3 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we present a comprehensive literature review of energy measurement approaches and compare our test bench with those approaches. In addition, various techniques to save energy costs of data centers and existing research on the energy efficiency of software applications are discussed. System model, implementation details and usefulness of automated test bench for energy measurement of software applications have been explained in Chapter 3. In Chapter 4, we show how developer choices can impact the energy costs of servers by choosing energy efficient APIs with the optimal choice of parameters while implementing file reading, file copy, file compression and decompression operations in Java. Chapter 5 presents a design of a framework which avoids repeated direct energy measurements for the common design options by making use of mathematical models of the energy costs. Some concluding remarks, limitations of the thesis and future work are provided in Chapter 6.

Chapter 2

Literature Review

2.1 Energy Measurement Approaches

Energy consumption measurement and estimation is a key requirement for providing better insight of how and where the energy is being spend in software. A deeper understanding of power costs of computing subsystems, namely, memory, processor, hard disk, and other peripherals, enables better use of storage encryption, virtualization, and application sandboxing [50][30]. The techniques for understanding the power cost of servers can be categorized into four major groups:

- *direct measurement* by means of instrumentation of the hardware [47]. Hardware measurement offers high precision but requires additional hardware whether embedded or not. In LEAP platform [47], additional hardware sensors are embedded in all the power rails to monitor the energy used by each hardware resource; whereas it is infeasible for anybody to do this in computer or server systems, except for the manufacturer of motherboard. Therefore, the main limitation of embedded approach is the inability for evolution and the difficulty to scale. Another way of direct measurement by measuring the current and voltage from the AC power supply to a whole computing system or to individual components without any modification to the underlying hardware. This later approach has been used by Zehan et. al in [11] to measure the power of main computer components with fine time granularity. They measured the current and voltage across all the wires of Advanced Technology Extended(ATX) power supply and figured out which wires power the main components, namely, CPU, hard disk and memory.

- *estimation* by means of power models [29][49]. Power models provide models to calculate or estimate the energy consumption of hardware and software. In power modelling, various software and hardware counters are used to predict power consumption after suitable training on an appropriately instrumented platform. The accuracy of modelling is the key for profiling the power. John et. al in [29] evaluated the effectiveness of model based power characterisation. They showed that linear regression modelling techniques work well only in restricted environment settings. Models are either too generic and course-grained [26][39], or platform dependent(in particular Java) [44]. They exhibit high prediction error in modern computing platforms due to many complexities such as multiple cores, hidden device states, and dynamic power components.

- *software measurement* by means of various tools and application programming interface (APIs). The main approach in software measurement is energy application profiling. Profilers help in understanding the system and decomposing the energy consumption of each hardware resource. They use software statistical sampling or software code instrumentation. Various profiling tools like PowerTop, Energy Checker, Joulemeter, ptop are discussed in detail in [37]. PowerScope does not offer energy information in real time unlike pTop. It first collect resources information at runtime and then calculates energy values of resources at a later stage of the measurement whereas ptop provides real time measurements and using them for dynamic energy-aware adaptations.

- *hybrid approaches* include frameworks composed of both software and hardware components. The hardware components include sensors, meters and data acquisition devices that enable direct power measurement and instrumentation. The software components include drivers for various meters and sensors, and user level APIs for controlling power profiling and code synchronization. We have also used this same approach in implementing an automated test bench for energy performance evaluation of software applications on servers. PowerPack [18] and pmlib software [6] are some of those frameworks, have automated the energy profiling of parallel scientific workloads by software code instrumentation. These tools have a set of user level APIs which one can insert before and after the code region of interest to create its energy profile. Both these tools did not talk about the applicability of their APIs to the target code of all programming languages. PowerPack requires additional sensing resistors for each of the power lines in addition to the power meter. Moreover, these tools can not be used to measure the energy cost of closed source applications. In contrast, our framework does not need the manual modification of source code of the

application for its energy measurement.

Although direct measurement of power consumption is expensive, it gives more accurate results than estimation models [29]. Our work also falls in the category of direct measurement.

In reference [37], a comprehensive survey of different energy measurement approaches has been done. Based on this survey, the authors have come up with four recommendations for the efficient energy measurement approaches:

- accurate measurements for better precision.
- fine-grained power models to trace how and where the energy is being used in software.
- reduce user experience impact - the measurement tools should not require manual modifications of source code of the applications. Approaches implementing energy models and formulas need to be invisible for the user, the application and the underlying system.
- software-centric approaches for better evolution and flexibility.

In another recent work [15], the authors designed a framework called software energy footprint lab, which executes the software of interest on the server and output the power consumed during the execution on a separate machine. Their approach requires manual effort to start the software under test and sending the commands to their Data Acquisition System right before the software is executed and another one right after it terminates, for synchronization. Our approach is different from them as PAST controls both the execution of the software as well as the measurement process. The process of synchronization between the server and the meter is automated in our approach. In addition, the measurement process of the same application can be repeated a number of times for statistical significance.

2.2 Techniques to save energy costs of Data Centers

To reduce the power bills of large data centers, researchers have proposed a wide range of energy saving techniques. These techniques include reducing cost at:

- hardware level by using power efficient cores, efficient memory and cache redesigning [22]. The "tickless idle" approach presented in [46] highlighted the limitation of CPU clock ticks happening irrespective of the status of the processor, i.e., regardless of it being busy or idle. The authors addressed this drawback by removing clock ticks in only idle processors. There is also a significant research work done in the area of high performance computing, for example assigning threads to a subset of the processors and enable sleep mode for unused processors while maintaining performance. Schall et. al in [43] enhanced the energy efficiency of database applications by using Solid State Drives(SSD) in place of conventional hard drives. They observed a significant drop in energy by using SSD. Mehta et al. in [31] developed a technique to reduce a processor's power consumption by reducing the power of the instruction registers. The number of memory operands can be reduced by using compilation policies that use the registers more effectively, which in turn lead to power savings, showed by Davidson and Jinturkar in [12].

- operating System level by: reducing the operating voltage and frequency for executing a particular task, known as Dynamic Voltage and frequency scaling (DVFS) [40], spin-down policies, adaptive placement of memory blocks, efficient use of component devices and energy aware routing [47]. The main focus here is to determine the policies that switch idle devices into low power states by predicting when the full capacity of the devices is not needed. The design of power aware operating system known as ECOSystem has been presented in [52]. In ECOSystem, power is managed as a system resource and is explicitly allocated to competing applications to achieve improvements in battery lifetime.

- power management level by decreasing the number of active servers to consolidate workload [54]. Workload consolidation is an effective way of saving power by turning off spare servers. This technique is mainly incorporated with virtual machines, which are migrated from many physical machines into a smaller number of physical machines.

- software level. Much research has been done to improve the energy efficiency of software at the compiler level by optimizing code to use fewer instructions or a more efficient ordering of instructions. Fraser et al. associate energy costs, with instruction patterns and then generate code through pattern matching using an algorithm that tries to find a cover of the pattern that minimizes the overall cost [17]. Research on the effects of application software activities on the energy consumption of servers is gaining momentum [9][36][48], because it is the software activities that drive the

operational cost of hardware, leading to varying dynamic power cost. Software applications designed with power cost in mind consume 40% less energy than other applications with the same functionality [48]. Nowadays there are many software applications intended to deliver the same functionality. For a particular task, user choice of applications has a great impact on energy consumption. Chenlei et al. [53] claimed that if users have the knowledge of different energy consumption behaviours among applications in the same category, they can choose the energy efficient ones among others that also provides the expected quality of service. They showed that by typing the document in gedit, then spellchecking it with Libre-Office, and finally uploading it to Google Docs can save much energy compared to doing all the tasks on Google Docs directly. Literature review of the existing research on the energy efficiency of software applications has been discussed in detail in next section.

2.3 Existing Research on the energy efficiency of software applications

Abdullah and Sanjeeda presented an energy efficient software development framework in [4]. Their framework defines a number of tasks that are grouped together into four different phases of the software development processes namely, develop, adopt, measure and optimize. The authors also presented several tools and techniques like Pipelining, Data parallelism and Task parallelism for improving performance and energy and related those techniques to different phases of the presented framework.

The authors of the paper [9] claimed that greater use of external libraries and application development environments would lead to higher energy cost of large scale applications. Their results showed that open source Java ERP system "Adempiere" (using Java Hibernate as an external library) consumes more energy than its counterpart "OpenBravo" (using plain SQL instructions), even though both the systems have same functionality. Therefore, developer's choice of API's and libraries in implementing a particular functionality has much impact on the energy cost of the software. The same authors in the another paper [8] experimentally showed that faster applications may also lead to higher CPU utilization which in turn can increase energy cost.

Ardito et al. in [5] developed the concept of introducing the energy efficiency into SQALE (Software Quality Assessment based on Lifecycle Expectations); one of the software quality models to monitor the impact of software on energy consumption during its development. They identified some energy efficient software guidelines and translated them

into measurable requirements of the model. Our work is different from them as we mainly focus on reducing energy cost in the coding stage of the software life-cycle.

Adel et al. in [14] observed that the energy cost of the same algorithm varies from language to language. They measured the energy cost of different implementations of Tower of Hanoi problem in C, C++, Java, Perl and Prolog and showed that implementation in C is most energy efficient among all. However, it is not practical for the developers to implement a feature in various programming languages and evaluate their energy cost, because it needs much expertise and IT companies might not afford this. We focus on exploring different ways of energy savings in a particular programming language.

High-level design and implementation choices of software engineers can play an important role in reducing the power consumption of application they code. The tools used by developers to measure the energy cost of their applications rarely exist. Cagri et. al in [42] developed a new tool for mapping software design to power consumption and describe how these mappings are useful for the software designers and developers in developing more energy efficient solutions. They studied in detail the impacts on energy usage on applying design patterns to an application. Their results showed that usage of design patterns can both increase and decrease the energy consumption of an application.

Hazem et. al in [19] presented a unique top-down approach for developing energy-aware software algorithms based on energy profiling. Their idea is to first identify and then measure the components of code with high energy consumption, known as kernels; which are frequently used operations in an algorithm. Their energy evaluation method involves isolated code with assembly injection.

Nattachart and Peraphon in [23] proposed a technique to reduce energy consumption of computer programs written in C using cohesion measure. Cohesion is a measure of how strongly related each piece of functionality expressed by the source code of a software module is. In other words, it refers to the degree to which the elements of a module belong together. The authors observed that higher the level of cohesion with in a program, the more power it consumed.

The impact of code refactoring techniques on the energy consumption of software has been studied by Jae-Jin et. al in [38]. Code refactoring techniques are meant to improve software performance, reliability as well as maintainability. The authors measured and analysed the power consumption of all the refactoring techniques developed by M Fowler [16]. Their results showed that among 63, only 33 techniques are energy efficient.

Various compression tools and compression formats on Linux servers have been compared in [28] as compression of data is a common operation in today's data centers. On

the other hand, in our work, we have compared the various IO, compression and decompression APIs available in Java programming language in terms of their energy cost. With the knowledge of energy behaviour of these APIs, developer can choose energy efficient APIs among others while implementing the file reading, file copy, file compression and decompression operations.

Chapter 3

Automated Test Bench for energy measurement of software applications

This chapter presents the design of an automated test bench to measure the power cost of an application running on a server. The test bench comprises a high precision power meter, a monitoring computer to acquire power readings from the power meter, and a newly developed control software (PAST) to synchronize the metering system with the load running on the server. We show how our test bench can be used by software developers to measure and improve the energy cost of two Java file access methods, namely, FileInputStream (M1) and BufferedInputStream (M2). Much energy can be saved by introducing a buffer of an appropriate size in both these methods. Another benefit of our test bench has been demonstrated by comparing the energy costs of compression and decompression features provided by two popular Linux packages: 7z and rar.

3.1 System Model of Test Bench

The system model of the automation framework has been shown in Figures 3.1 and 3.2. The definitions of all the terms used in the figures are given below.
Server: A system for which we are interested in evaluating the energy cost of running an application.
Load: A software application that runs on the server, and we measure the energy cost of running that application.
Power Meter: A data acquisition unit used for measuring power. We used a Lab-Volt

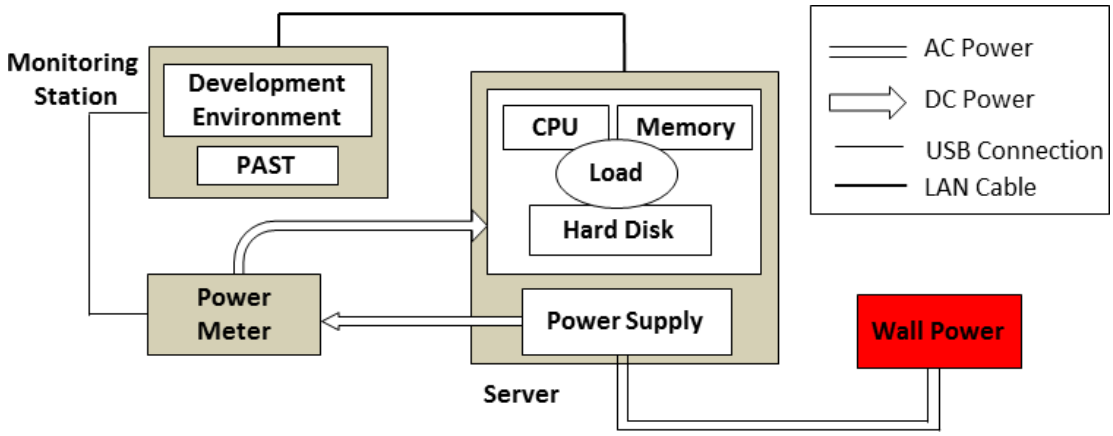

Figure 3.1: System Model for Desktop where power lines to Disk, Memory and CPU are identifiable

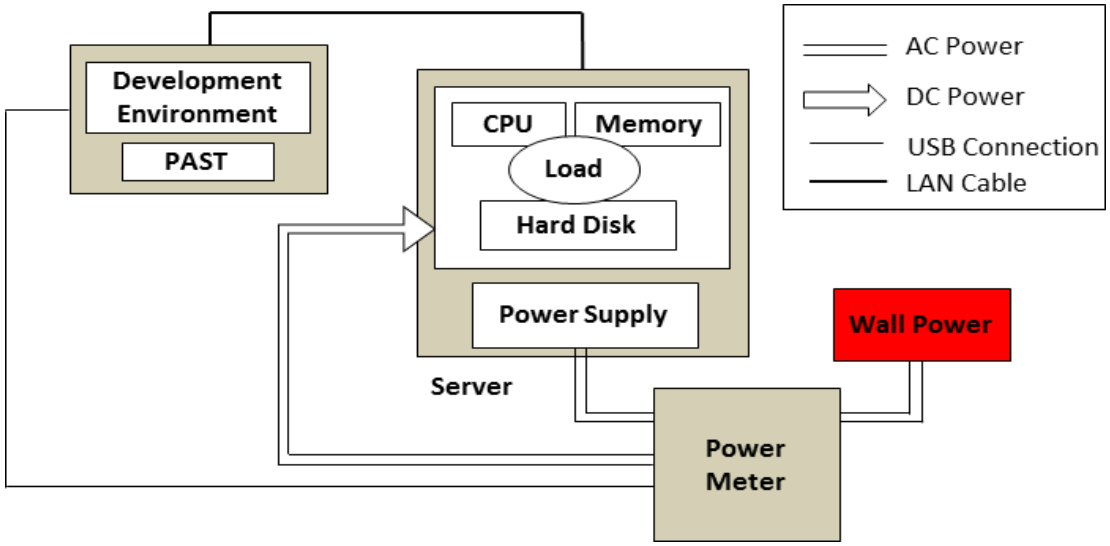

Figure 3.2: System Model for Server where power lines to individual subsystems are not identifiable

9063-00 Data Acquisition and Control Interface as a power meter.
Wall Power: Supplies AC power to the Server.
Monitoring Station: A computer equipped with the PAST system which controls both the Server and Meter. A programmer is developing the application on this machine and can run PAST to upload the application to the Server and measure its cost. By running PAST on a separate machine, it starts executing the Load on the Server as well as starts the Meter to record current and voltage values simultaneously.

The Monitoring Station is connected to the Meter via an USB (Universal Serial Bus) interface and to the Server through a LAN (Local Area Network).

Our test bench can be used to measure:

- the power consumed by a server's individual subsystems, namely, memory, disk, and processor, if their power lines are easily accessible (Figure 3.1); and
- the total power cost of a server. Only the total power can be measured for a server where one cannot identify the power lines to its individual subsystems (Figure 3.2).

To set up the test bench for power measurement of individual subsystems, we examined the different power lines from the ATX 24 pin connector which powers the whole motherboard of desktop computer. The power lines to the processor (CPU: central processing unit) operate at $12V$, first fed to the voltage regulator module which converts the voltage to the actual voltage required by the processor [24]. From the 24 wires of ATX connector, one yellow wire of $12V$ is feeding power to the processor. The other two yellow wires are from the ATX 4 pin $12V$ Power Connector (ATX v2.2) dedicated for the processor. The disk (Hard Disk) is getting power from a Molex 4 pin power supply connector which operates at two voltage levels, $5V$ and $12V$. The memory (RAM: random access memory) system is getting power over three lines from the 24 pin connector, and the voltage level is $3.3V$. The total power cost can be measured from the AC (Alternating Current) power lines to the server power supply.

If one is interested in measuring the power cost of individual subsystems, namely, processor, memory, and disk, it is important to identify the number of separate power lines to monitor for two reasons:

1. data acquisition systems come with a small number of input channels.
2. each subsystem of a computer receives power over multiple power lines. For example, the data acquisition system that we used in our test automation has $four$ power input

channels, and the memory subsystem of the desktop computer alone, that we used for measurement, has three power lines.

Therefore, measuring the total power of a server is easier than measuring the power for individual subsystems.

3.2 Automated Test Bench

In our test bench, we use a Lab-Volt 9063-00 Data Acquisition and Control Interface system, known as the Meter in this paper. To read power samples from Meter, the device supports APIs in the form of Microsoft Dynamic Link Library (DLL). Therefore, the PAST is developed in Visual Basic. In the remainder of this section, we explain the design of PAST by means of its behaviour, which is represented as a message sequence chart then we explain the key problems faced in the design of the PAST, and describe the PAST in pseudocode form.

3.2.1 Message Sequence Chart

Figure 3.3 shows the sequence of steps of the PAST executed during the whole process of measurement.

PAST is a multi-threaded system, with three threads: MainThread, LoadThread and MeterThread. The PAST is launched on the Monitoring station with the location of the configuration file as its input parameter. A configuration file is a text file that is stored on the Monitoring Station, it contains both the Server and Meter information. Figure 3.4 shows some entries from configuration file. The behaviours of the three threads is described below:

MainThread: MainThread first reads the configuration file for the *server_ipaddress*, *username*, *password* and the location on the server (*server_app_loc*) where the developer wants to upload the application. *Launch_app_command* contains the command to start an application on the server. *meter_inputs* in the configuration file tells which current and voltage input channels of the Meter and at what sampling frequency (*meter_sampling_freq*) the Meter should produce those values. It then starts the LoadThread and waits for the other threads to finish.

LoadThread: This first uploads the application onto the server. It stops the process if the application is not uploaded successfully. If upload is successful, then it initializes the Meter with the meter information being read from the configuration file. If the Meter is ready to read then it starts a new thread MeterThread and starts the application on the Server.

MeterThread: This starts recording the current and voltage values from the Meter by using meter API calls. LoadThread ensures that the MeterThread is recording the values till the application is running on the server. And finally it saves all the values in to the file inside directory (*Recording_dir*) on the Monitoring Station.

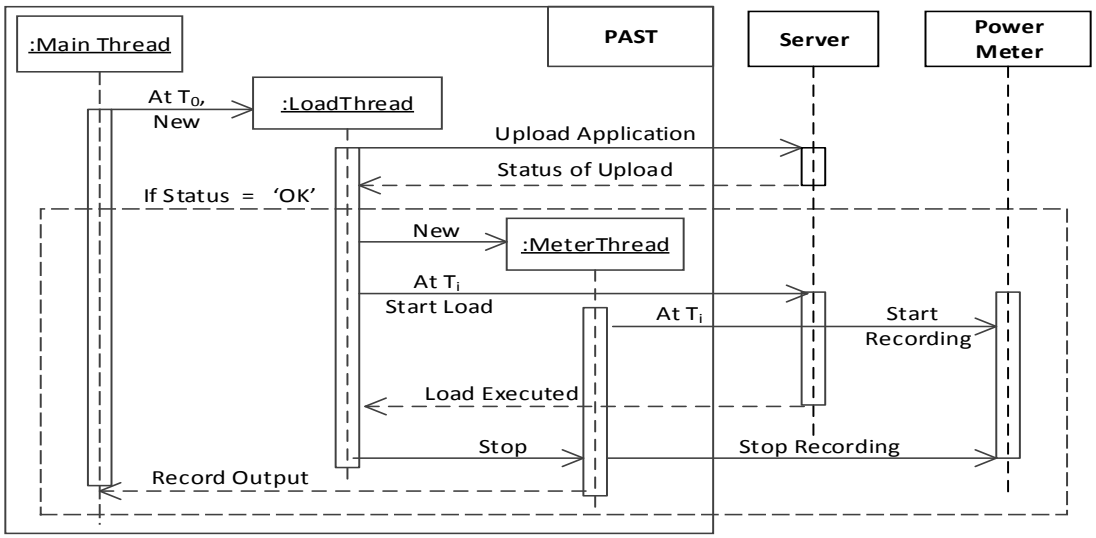

Figure 3.3: Message Sequence Chart

From the recorded current and voltage values, energy cost of running an application on Server is computed by using the expression

$$Energy\_cost = \sum_{\forall i} V(i).I(i).\Delta t$$

where $V(i)$ and $I(i)$ are the i^{th} voltage and current samples, respectively, and Δt is the sampling interval.

```
server_ipaddress=192.168.1.148
username=developer
password=developer
local_application=C:\MyApp.jar
server_app_loc=/home/jasmeet/
Launch_app_command=java - jar MyApp.jar
iterations=5
component=CPU
tunable_parameter=BufferSize
tunable_parameter_array=128,256....
Recording_dir=C:\Power\Results
server=linux
meter_sampling_freq=1000
meter_inputs=E1,I1,I2,I3
```

Figure 3.4: Sample Configuration file

3.2.2 Key Challenges

There are some practical problems in measuring the energy cost of an application at the subsystem level, namely, processor, memory, and hard disk. There are only 4 current and voltage inputs to the meter. Therefore, at a time only 4 power channels can be measured. However, in case of our desktop, for all the three components (processor, memory and hard disk), there are a total of 8 power lines needed to be monitored. Therefore, we measure the power cost of the three subsystems in three repeated experiments.

3.2.3 Pseudocode of PAST

Algorithm 1 illustrates the pseudocode of PAST. Line #1-5 declare all the variables meant to store the Server and Meter information. Two thread objects, LoadThread and MeterThread are declared initially and then later the job of executing the Load and measurement are assigned to them respectively. An object $Load\_Proc$ of type process is also declared to create a new process which is responsible for starting the Load.

MainThread: (Line #6-24) The location (loc) of the configuration file is given as an input argument to the MainThread. Configuration file is a text file that is stored on the Monitoring Station, contains both the Server and Meter information. Variables $IPADDRESS, PASS, SERVER\_PLAT$ of configuration file corresponds to the ipaddress, username and password of the Server respectively. APP is application used as a Load on Server and $PARAM\_ARRAY$ is the tunable parameter array. This array contains

all the values of the tunable parameter for which we want to run the application. Parameter $METER\_INPUTS$ tells which current and voltage input channels of the Meter to read and $SAMPL\_FREQ$ is sampling frequency at which the Meter is producing those values. If the configuration file does not exist on the location specified, MainThread stops the PAST after raising an Exception. If the file exists, it first reads both the information and stored in to variables that are accessible to the other two threads. It then starts the LoadThread and waiting for the other threads to finish.

LoadThread: (Line #25-46) It first checks the Server Platform and sets up the new Windows Process (*Load_Proc*) information differently for the Linux and Windows Server platform. Two windows utilities, psexec and plink that are used by *Load_Proc* to execute the Load on the Windows and Linux servers respectively. Then it initializes Meter with the sampling frequency and current and voltage input channels to read by calling a function InitializeMeter(). If the Meter is ready to read then it starts a new thread MeterThread and parallely starts the process *Load_Proc*. *Load_Proc* stops when the Load finish executing on the Server.

INITIALIZEMETER(): (Line #47-55) It contains all the calls to the Power Meter API. initDevice() checks that the Power Meter is connected to the Monitoring Station and initializes it to read the votlage and current channels specified in $METER\_INPUTS$. It returns 0 if the meter is properly initialized, otherwise return -1.

MeterThread: (Line #56-65) It starts recording the current and voltage values from the Meter by using meter API calls. LoadThread ensures that the MeterThread is recording the values till the *Load_Proc* is running through a common variable $RECORDING$. LoadThread sets $RECORDING$ to $true$ while the *Load_Proc* is running and make it $false$ when it finishes and MeterThread is read the values while the $RECORDING$ is $true$. And finally save all the values in to the file inside directory specified in $RECORD\_DIR$ on the Monitoring Station.

This complete process of executing the Load and recording the current and voltage values is done for each value of the tunable parameter array. And for each value, this process is repeated for the number of times equal to $ITERATIONS$ parameter specified in the Configuration File.

Algorithm 1 PAST

```
1: string ITERATIONS, RECORD_DIR, SERVER_PLAT
2: string METER_INPUTS, SAMPLING_FREQ
3: string APP_ATTR[4]
4: LoadThread, MeterThread ← New Thread
5: Load_Proc ← New Process
6: function MainThread(loc)
7:     CONF_FILE ← loc
8:     if exists(CONF_FILE) = true then
9:         IPADDR, APP ← read(CONF_FILE)
10:        UNAME, PASS ← read(CONF_FILE)
11:        PARAM_ARRAY ← read(CONF_FILE)
12:        ITERATIONS ← read(CONF_FILE)
13:        RECORD_DIR ← read(CONF_FILE)
14:        SERVER_PLAT ← read(CONF_FILE)
15:        METER_INPUTS ← read(CONF_FILE)
16:        SAMPL_FREQ ← read(CONF_FILE)
17:        APP_ATTR = {IPADDR, UNAME, PASS, APP}
18:        StartMeasurement ← LoadThread.job()
19:        MainThread.wait()
20:        PRINT "Measurement Process Done"
21:    else
22:        EXCEPTION "No Configuration file"
23:    end if
24: end function
25: function startMeasurement
26:    if SERVER_PLAT = Windows then
27:        Load_Proc.info(psexec,APP_PARAMS)
28:    else
29:        Load_Proc.info(plink,APP_PARAMS)
30:    end if
31:    for each PARAM in PARAM_ARRAY do
32:        for i ← 1, ITERATIONS do
33:            STATUS ← InitializeMeter()
34:            if STATUS = 0 then
35:                StartRecording ← MeterThread.job()
36:                Load_Proc.start()
```

```
37:                 while !Load_Proc exited do
38:                     RECORDING ← true
39:                 end while
40:                 RECORDING ← false
41:                 LoadThread.wait()
42:             end if
43:         end for
44:     end for
45:     MainThread.notify()
46: end function
47: function InitializeMeter
48:     if meter.initDevice() = 0 then
49:         meter.set(METER_INPUTS)
50:         meter.set(SAMPL_FREQ)
51:         return 0
52:     else
53:         return −1
54:     end if
55: end function
56: function StartRecording
57:     LIST ← new ArrayList
58:     while RECORDING = true do
59:         TEMP ← readIandV(METER_INPUTS)
60:         LIST.add(TEMPARRAY)
61:     end while
62:     meter.closeDevice()
63:     write(LIST,RECORD_DIR)
64:     LoadThread.notify()
65: end function
```

3.3 Experiments

In this section, we show how software developers can use our test bench to evaluate the energy performance of running an application on a server with various design options. We compare the energy cost of two Java file access methods: (i) M1 using FileInputStream only and (ii) M2 using BufferedInputStream. Ardito et al. [5] intuitively claim about the energy efficiency of these two methods without any measurements. First, we validate their claim by measuring the energy cost of the methods on our test bench. Then we revise the two methods by introducing a *buffer* into them and measure their energy cost with varying buffer sizes. We also compare the revised methods to read extremely large files in terms of

Table 3.1: Server Machines Configuration

Parameter	Desktop (ASUS P4P800-VM)	Real Server (Dell PowerEdge 2950)
Processor	Intel Pentium 4, 3.2 GHz	7x Intel Xeon, 3 GHz, 4 cores per processor
Hard Disk	80 GB IDE	1.7 Tera Bytes SAS
Main Memory	2 GB DIMM	32 GB DIMM
Operating System	Linux (Ubuntu 13.10)	Linux (Ubuntu 13.10)

their energy cost. Next, we compare the energy performance of two packages *7z* and *rar* with respect to compression and decompression. Table 3.1 shows the configurations of two machines used in our experiments.

3.3.1 Example of using test bench to make important design decisions

Listing 1 and Listing 2 in Figure 3.5 describe M1 and M2, respectively. We measure the energy cost of CPU, memory and disk for reading a video file of size 512 MB (Mega Bytes) with M1 and M2 on a desktop machine. Figure 3.5 shows the results of our measurements by comparing the energy cost of all the three components for both M1 and M2. The reason behind the less energy consumption by M2, for all the components is that it reads a file

```
FileInputStream fis = new FileInputStream(fileName);
int b,cnt = 0;
while ((b = fis.read()) != -1)
{
        if (b == '\n')
                cnt++;
}
fis.close();
```

Listing 1. **M1**: File Reading using FileInputStream

```
FileInputStream fis = new FileInputStream(fileName);
BufferedInputStream bis = new BufferedInputStream(fis);
int b,cnt = 0;
while ((b = bis.read()) != -1)
{
        if (b == '\n')
                cnt++;
}
fis.close();
```

Listing 2. **M2**: File Reading using BufferedInputStream

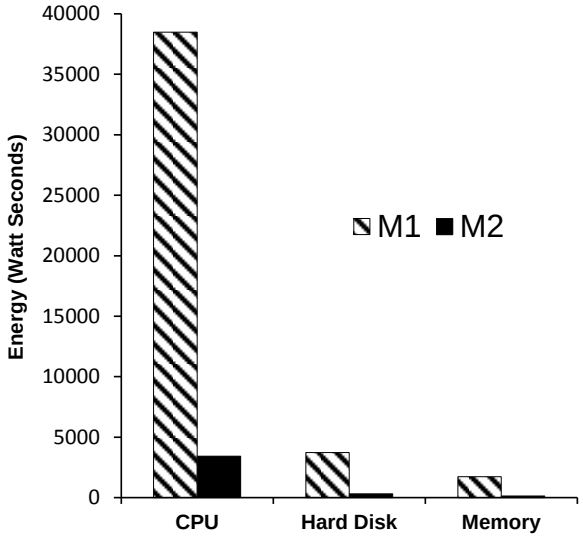

Figure 3.5: Energy cost evaluation of CPU, Memory and Disk for M1 and M2 on the Desktop

```
1 FileInputStream fis = new
      FileInputStream(fileName);
2 byte[] buffer = new byte[bufferSize];
3 int b,cnt = 0;
4 while ((b = fis.read( buffer )) != 1)
5 {
6         if (b == '\n')
7                 cnt++;
8 }
9 fis.close();
```

Listing 3: **M1'**: Introducing user buffer in M1

```
 1 FileInputStream bis = new
       FileInputStream(fileName);
 2 byte[] buffer = new byte[bufferSize];
 3 BufferedInputStream bis = new
       BufferedInputStream(fis);
 4 int b,cnt = 0;
 5 while ((b = fis.read( buffer )) != 1)
 6 {
 7         if (b == '\n')
 8                 cnt++;
 9 }
10 fis.close();
```

Listing 4: **M2'**: Introducing user buffer in M2

of any size in larger chunks equal to the size of its internal buffer from the disk, whereas M1 reads a single byte of data in one read operation. It is clear from the results that CPU consumes the maximum energy in reading a file.

We further study the impact of introducing a programmer defined *buffer* into both the

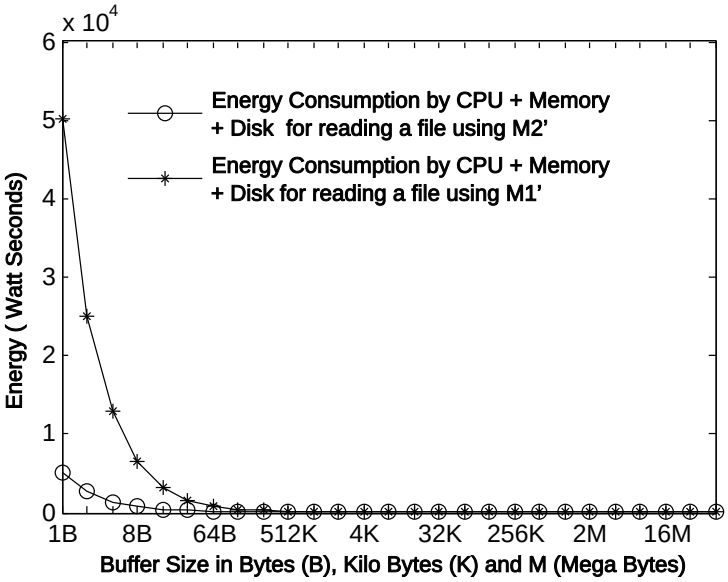

Figure 3.6: Total energy cost by M1' and M2' with different buffer sizes on the Desktop

methods. Listing 3 and Listing 4 describe the modified code of the two methods, and they are denoted by M1' and M2' corresponding to M1 and M2, respectively. In both M1' and M2', line #2 shows the definition of *buffer* as an array of type byte, and its size is equal to *bufferSize*. Line #4 and #5 of M1' and M2' respectively, show that in one call *read* operation reads several bytes of data of size, *bufferSize*. Therefore *bufferSize* is a tunable parameter which the developer can vary and run these methods to read a file. We measure the energy cost of CPU, memory and Disk for both M1' and M2' with buffer size ranging from 1 Byte to 64 Mega Bytes (MB).

Figure 3.6 shows the evaluation of the total energy cost of all the three components for both M1' and M2'. The results in Figure 3.6 show that after introducing a programmer buffer into M1 and M2, the total energy cost of all the three components, is maximum at buffer size 1 byte. It started decreasing with the increase in the buffer size till 128 bytes. We expanded the graphs of Figure 3.6 in Figure 3.7 to show the energy cost of individual components along with their total energy cost at the buffer sizes from 128 bytes to 64 MB.

Figures 3.7(a), 3.7(b) and 3.7(c) show the energy cost of CPU, memory and disk respectively, and their total energy cost in 3.7(d). The energy cost behaviour between M1' and M2' is same as between M1 and M2 for buffer sizes from 128 bytes to 8KB; in other words, energy cost of M2' remains less than M1'. Then, energy is constant for both the methods ranging from 8KB to 128KB, except that there is a sharp increase at 32KB by M1' for

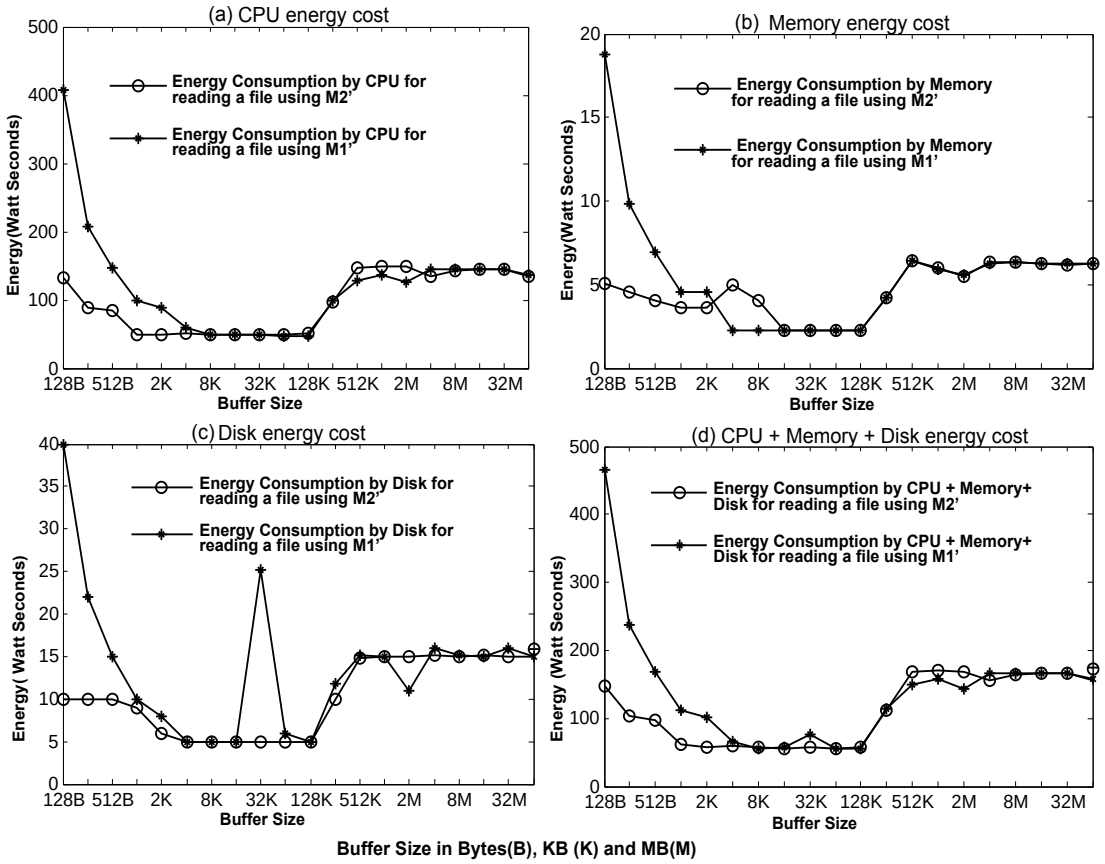

Figure 3.7: Energy cost of M1' and M2' with different buffer sizes on the Desktop

disk. It started increasing from 128KB to 1 MB then decreases and remains constant till 64MB. Both M1' and M2' consume almost the same energy for all the three components from 8KB to 64MB and consumes minimum energy at 16KB. Moreover, this energy is even less than M2.

Therefore, it is clear from our measurements that there is a further opportunity to decrease the energy cost of M1 and M2 by introducing a programmer buffer into them. Both the methods consume almost the same energy at buffer sizes ranging from 8K to 64 MB which contradicts the claim by Ardito et. al [5] that M1 always consumes more energy than M2. In addition to this, 16K is the optimal buffer size for all the three components.

To gain additional insights into the behaviours of M1' and M2' while reading extremely large files, we perform the experiments on the same desktop machine to read files ranging

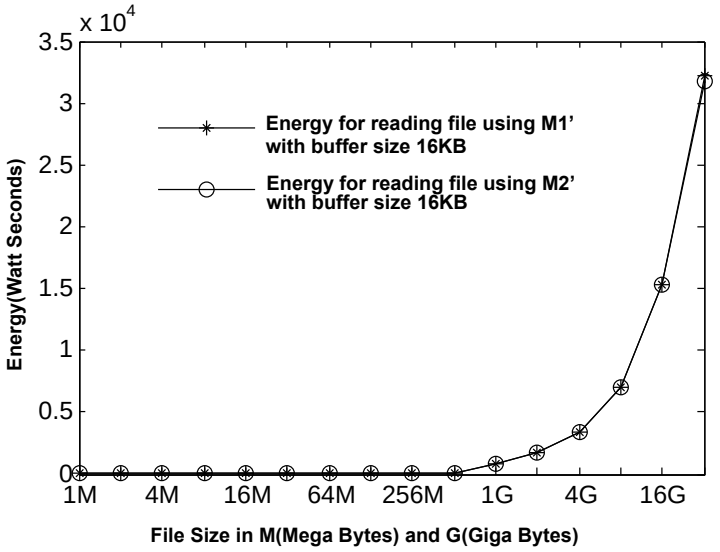

Figure 3.8: Energy cost of M1' and M2' for different file sizes on the Desktop

from 1MB to 32 Giga Bytes(GB), while keeping the buffer size fixed at 16KB. Figure 3.8 shows the graphs plotted for the total AC (Alternating Current) energy cost as function of different file sizes for both the methods at 16KB buffer size. It is clear from the graph that both the methods consume the same energy at 16KB buffer size. The initial portion of Figure 3.8 has been zoomed in Figure 3.9 for the file size ranging from 1MB to 512 MB. The energy cost gradually increases for the file sizes from 1MB to 512MB but there is a significant increase in energy cost from 512MB to 32 GB. The sharp rise in energy for large file sizes needs to be further investigated on a server with large memory. We close this section by noting the above results enable the developer to chose the right method for reading a file with appropriate buffer size during the design stage.

3.3.2 Using the test bench in function level energy cost measurement

The test bench can also be used to measure the energy cost of a specific function of an application software whether it is open source or closed source. To validate this functionality of our test bench we conducted the experiments on a real server (Table 3.1) from a data center. We consider two popular compression packages, namely, *7z* and *rar* to compress and decompress files on both Linux and Windows operating systems but same hardware platform. Both the packages output compressed files in *.rar*, *.7z* and *.zip* formats and can

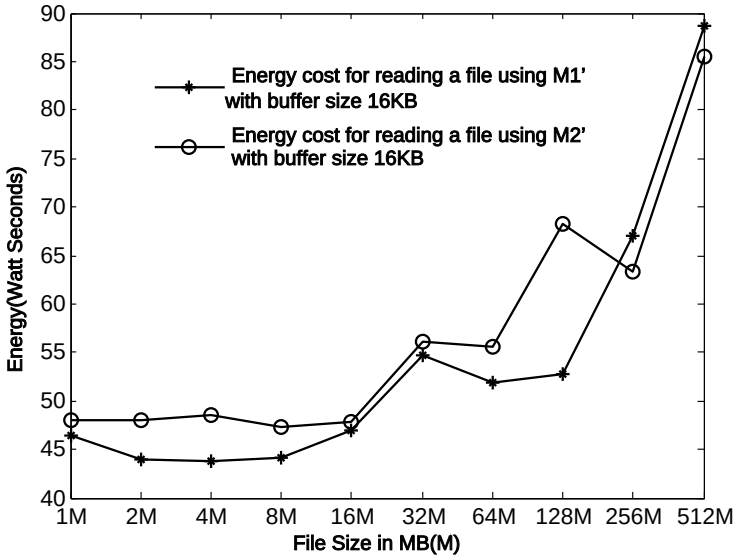

Figure 3.9: Energy cost of M1' and M2' for different file sizes on the Desktop

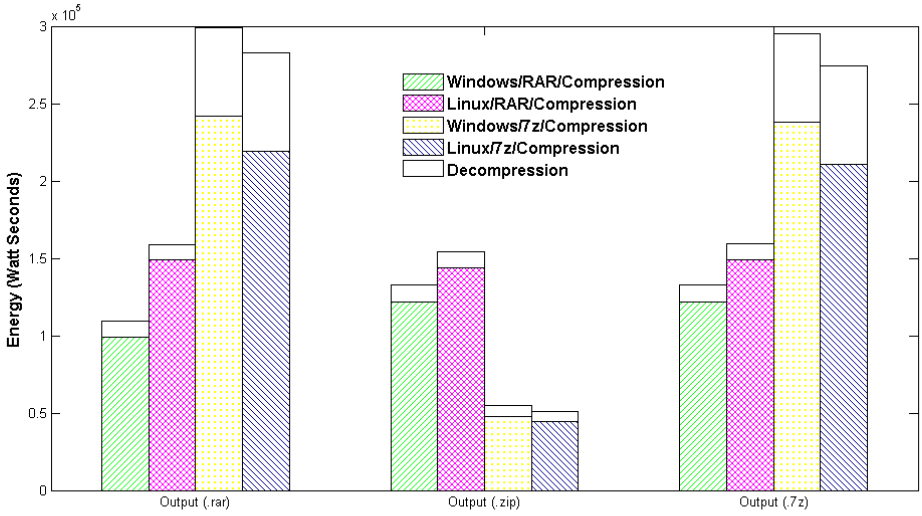

Figure 3.10: Energy cost of compression and decompression functions of *rar* and *7z* packages on the real server (Table 3.1)

decompress the same to the original files. Figures 3.10, 3.11, 3.12 shows comparison of various combinations of OS (Windows and Linux), Compression packages (*rar* and *7z*), and output formats (*.rar*, *.zip*, and *.7z*) for 2 GB mp4, 370 MB pdf, 370 MB excel files respectively. The entries within the parentheses on the X-axis represent the format of the output produced by the compression packages. Each bar represents the energy cost of compression and decompression performed on a respective files on a Dell server.

The measurement results show that:

- For the three types of files (.mp4, pdf and .xlsx), both the packages behave similar in terms of energy cost on Linux and Windows OS.
- *rar* package consumes less energy on Windows as compare to on Linux in producing all the three formats (*.rar*, *.zip* and *.7z*). On the other hand using *7z* on Linux is more energy efficient than using it on Windows.
- *rar* package consumes same energy in producing three formats whereas *7z* consumes minimum energy in producing *.zip* format.
- Overall, *7z* on both Linux and Windows, is most energy efficient in compressing the files to *.zip* format as compare to other output formats.

Further investigation is required to find the causes of energy cost differences of the same operations of two packages.

3.4 Summary

In this paper we presented an automation framework to measure the energy cost of servers while running software applications. The framework's infrastructure mainly contains a power meter, target server and control software (PAST) for synchronization and monitoring. By using the test bench, we performed actual measurements to verify the claim in a previously published paper [5] that energy cost of reading files by the method FileInputStream (M1) is greater than the BufferedInputStream (M2) method. However this claim is not valid in certain cases, if we introduce a programmer buffer in both the methods. It holds good for buffer sizes ranging from 128 bytes till 8KB, but these two methods consume almost the same energy at buffer sizes from 8KB to 64MB. Also, the introduction of *buffer* in M2 has further reduced its energy cost. Finally, we compared the energy costs

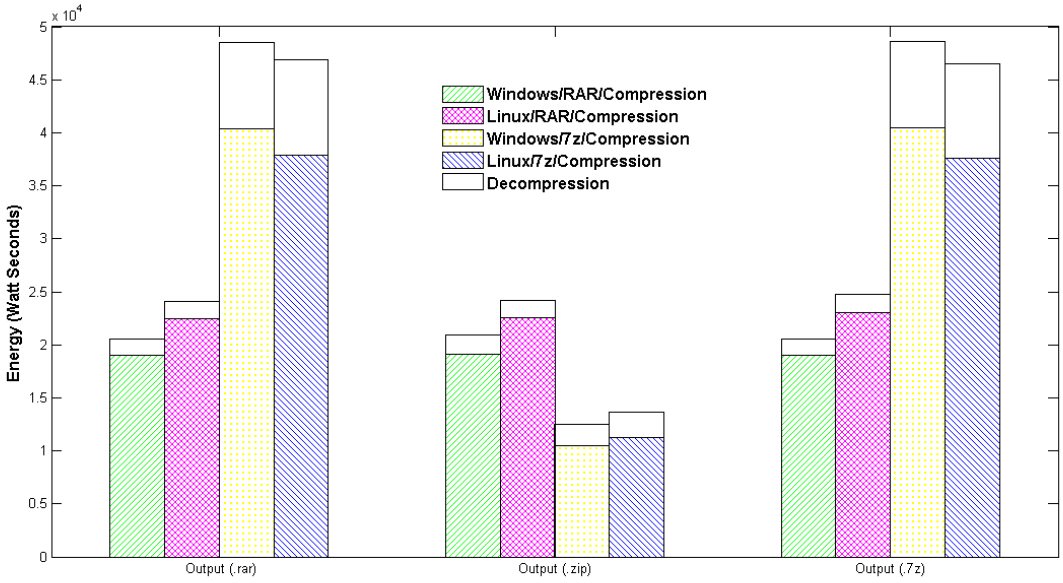

Figure 3.11: Energy cost of compression and decompression functions of *rar* and *7z* packages on the real server (Table 3.1)

of the same functionality provided by different software applications by measuring the energy costs of compression and decompression features of two Linux packages: *7z* and *rar*. The *7z* package consumes more energy than rar in compressing and decompressing files. The automation framework can be used by programmers to evaluate the energy cost of their applications. More work is required to be done to find out why the energy cost rises significantly for extremely large file sizes (Figure 3.8)

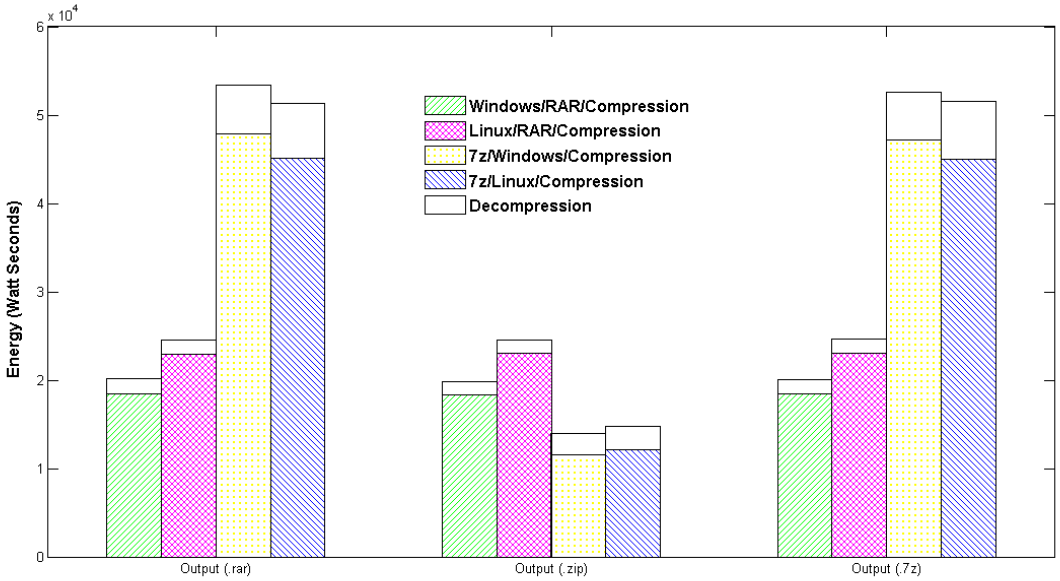

Figure 3.12: Energy cost of compression and decompression functions of *rar* and *7z* packages on the real server (Table 3.1)

Chapter 4

Impact of developer choices on energy consumption of software applications

With the growth in cloud computing, data-intensive computing has emerged which involves processing large volumes of data, commonly referred to as big data. The increasing demand for high performance reading and writing of big data leads to the high usage of computing resources in data centers which results in high power bills. To analyze and process big data, various big data and cluster computing frameworks have been proposed in the past. Those frameworks include Apache Hadoop, Pregrine, Apache Spark and Dryad. All these frameworks have been integrated with most of the high level programming languages namely, Java, Python, and C# for performing input/output (IO) operations. These IO operations mainly include reading big data, transferring huge data from one server to another, compressing data to gain storage space, and decompressing the data to use it. Within a particular programming language, there are various methods and Application Programming Interface (APIs) available to carry out these operations. If developers are aware of the power consumption of these APIs, they can choose energy efficient APIs and their optimal parameters while implementing these operations.

In this Chapter, we analyze the various APIs available in Java programming language to implement common operations, namely, file reading, file copy, file compression and file decompression on large servers from the energy viewpoint. There are some APIs in Java which have a tunable parameter, *buffer size*, which developers can change according to their choice and use the APIs to perform those operations. By measuring the energy cost

of the tunable APIs with different buffer sizes, we show much energy can be saved by choosing a particular *buffer size*. For the comparison, we measure the energy cost of those APIs which do not have any tunable parameter. Specifically, we evaluate the following scenarios on both Linux and Windows servers running on the same hardware platform:

- file reading with three APIs, namely, traditional FileInputStream and BufferedInputStream and the latest Java 7, FileChannel with different buffer sizes.
- file copy by Streams (FileOutputStream and BufferedOutputStream) with different buffer sizes.
- comparison of minimum and maximum energy costs of file copy by using Streams at a particular *buffer size* with the energy cost of other available copy methods including Java 7 Files.copy method, Apache FileUtils.copyFile and FileChannel transfer method in which buffer size is fixed.
- file compression in *zip*, *gzip* and *xz* formats and their decompression by using Java APIs.

4.1 Methodology

We have used the same automated test bench presented in Chapter 3, for measuring the energy cost of an application running on a sever. We measure the total AC (Alternating Current) power cost of a server as the power lines to individual subsystems, namely, hard disk, CPU and memory cannot be identified on commercial large scale servers.

4.1.1 Choosing key operations on Servers

Key operations on Cloud based software systems in data centers include:

- Reading and processing huge data from files. Several thousands of Hadoop job instances are required for reading a big database index file to memory, and do a run-time look up to base the index. Therefore, energy efficient IO is required to reduce energy cost.[45][13].
- Transfer (Copy) big files from one server to another.

- Data compression to save disk space and to improve performance which in turn saves energy [28].
- Data decompression to process and analyse the compressed data.

In this study, we choose to explore various APIs and methods available in Java to carry out these operations. We analyzed these operations from the energy viewpoint by reading, copying, compressing and decompressing large files on a large scale server by using the appropriate APIs in Java. Java is chosen because the most commonly used big data frameworks are either written in Java or they have the support for Java to perform IO operations, data compression and decompression.

4.1.2 Various Input/Output API's in Java for File Reading and File Copying

In Java programming, IO has been carried out using Streams (standard input/output) until new input/output (NIO) library (block oriented) was introduced with JDK 1.4. The most important distinction between the original IO library (found in java.io.*) and NIO has to do with how data is packaged and transmitted. Following are the main points which highlight the differences between the traditional IO and NIO in Java:

1. A stream-oriented IO system deals with the movement of data one or more bytes at a time, through an object called a Stream. A single byte or multiple bytes of data can be explicitly read from or written to streams by a programmer defined byte array. This array can be called a buffer which improves the energy efficiency by reducing the disk access operations, whereas block-oriented IO system deals with data in blocks. Each operation produces or consumes a block of data in one step. Instead of streams, all data that goes from anywhere (or comes from anywhere) must pass through a Channel object. By default, all data is handled with buffers. But this buffer is more than just an array; its contents occupy the same physical memory used by the underlying operating system for its native IO operations, thus enabling the most direct transfer mechanism and eliminating the need for any additional copying.

2. Java IO's various streams support blocking operations. In other words, when a thread invokes a read() or write(), that thread is blocked until there is some data to be read or the data is fully written. On the other hand, in asynchronous NIO, a thread can

Table 4.1: Various File Reading APIs in Java

I/O System	API(s)	Notation	API Usage in a Program
java.io.*	FileInputStream.read()	M1	1 FileInputStream fis = 2 new FileInputStream(fileName); 3 byte[] **buffer** = new byte[**bufferSize**]; 4 int b,cnt = 0; 5 while ((b = fis.read(**buffer**)) != 1) 6 { 7 cnt++; 8 } 9 fis.close();
java.io.*	BufferedInputStream.read()	M2	1 InputStream fis = 2 new FileInputStream(fileName); 3 byte[] **buffer** = new byte[**bufferSize**]; 4 BufferedInputStream bis = 5 new BufferedInputStream(fis); 6 int b,cnt = 0; 7 while ((b = bis.read(**buffer**)) != 1) 8 { 9 cnt++; 10 } 11 fis.close();
java.nio.*	FileChannel.read()	M3	1 RandomAccessFile file = new 2 RandomAccessFile(fileName, "r"); 3 FileChannel inChannel = 4 file.getChannel(); 5 ByteBuffer **buffer** = 6 ByteBuffer.allocate(**bufferSize**); 7 while(inChannel.read(**buffer**) > 0) 8 { 9 **buffer**.flip(); 10 **buffer**.clear(); 11 } 12 inChannel.close(); 13 file.close();

request that some data be written to a Channel, but not wait for it to be fully written. In the meantime, the thread can proceed on performing IO on other channels; i.e. a single thread can now manage multiple channels of input and output.

3. NIO moves the most time-consuming I/O activities (namely, filling and draining buffers) back into the operating system, thus allowing for a increased speed.

Tables 4.1 and 4.2 shows the various APIs available in Java based on standard IO and NIO system for file reading and file copy operations, respectively. The third column of the tables shows the main API(s) being used to perform an operation, the fourth column contains the notation that we use to represent the usage of these APIs by means of Java methods, and the fifth column (API usage in a Program) contains the actual implementation of the programs to carry out file reading and file copy operations. All M1, M2 (from standard Java IO) and M3 (from NIO) can read a large file in chunks (*buffer*) of size equal to *bufferSize* but *buffer* in M1 and M2 is different from M3's buffer as explained in the first paragraph. Line #2 of M1 and M2 and Line #4 of M3 from the fifth column of Table 4.1 show the definition of *buffer*. Line #4, #6 and #5 of M1, M2 and M3, respectively, show that in one call, *read* operation reads several bytes of data of size *bufferSize*. Therefore, in all file reading methods (M1, M2 and M3), *bufferSize* is a tunable parameter which developers can change according to their choice.

The file copy methods MC1 and MC2 in Table 4.2 also use the same APIs as M1 and M2, respectively. On the other hand, MC3, MC4 and MC5 do not have any parameter which can be varied in contrast to MC1 and MC2 where the size of the *buffer* can be defined by a developer. In addition, MC3 is a part of NIO introduced in JDK 1.7; it uses platform's File System providers to copy files from one location to another. MC5 uses NIO file Channels to transfer files. Further, Apache FileUtils library method is used in MC4 which in turn uses NIO file Channels similar to that of MC5. This library method has a simpler interface for file copy operation as compare to that of file Channel.

Table 4.2: Various File Copy APIs in Java

I/O System	**API(s)**	**Notation**	**API Usage in a Program**
java.io.*	FileInputStream.read() FileOutputStream.write()	MC1	1 File source = new File(srcFile); 2 File dest = new File(destFile); 3 OutputStream os = 4 new FileOutputStream(dest); 5 InputStream is = 6 new FileInputStream(source); 7 byte[] **buffer** = 8 new byte[**bufferSize**]; 9 int length; 10 while((length=is.read(**buffer**))>0) 11 { 12 os.write(**buffer**, 0, length); 13 }
java.io.*	BufferedInputStream.read() BufferedOutputStream.write()	MC2	1 File source = new File(srcFile); 2 File dest = new File(destFile); 3 OutputStream dst = 4 new FileOutputStream(dest); 5 InputStream src = 6 new FileInputStream(source); 7 BufferedOutputStream os = 8 new BufferedOutputStream(dst); 9 BufferedInputStream is = 10 new BufferedInputStream(src); 11 byte[] **buffer** = new byte[**bufferSize**]; 12 int length; 13 while ((length = is.read(**buffer**)) > 0) 14 { 15 os.write(**buffer**, 0, length); 16 }
java.io.*	Files.copy()	MC3	1 File source = new File(srcFile); 2 File dest = new File(destFile); 3 Files.copy(source.toPath(), 4 dest.toPath());

Continued on next page

Table 4.2 – *Continued from previous page*

I/O System	**API(s)**	**Notation**	**API Usage in a Program**
java.io.*	apache.FileUtils.copyFile()	MC4	1 import apache.commons.io.FileUtils 2 File source = new File(srcFile); 3 File dest = new File(destFile); 4 FileUtils.copyFile(source,dest);
java.nio.*	FileChannel.transferFrom()	MC5	1 File src = new File(srcFile); 2 File dest = new File(destFile); 3 FileChannel srcChan = new 4 FileInputStream(src).getChannel(); 5 FileChannel destChan = new 6 FileOutputStream(dest).getChannel(); 7 destChan.transferFrom(srcChan, 8 0,srcChan.size());

4.1.3 Data Compression and Decompression APIs in Java

The Java platform has mainly two APIs for compressing data: (i) ZipOutputStream writes the data out in a compressed *zip* format; and (ii) GZipOutputStream compresses data in the *gzip* format. Both the APIs are part of java.util.zip package and use *Deflate* as a data compression algorithm. The *Deflate* algorithm is a combination of the LZ77 algorithm and Huffman coding. The third compression API that we studied is XZOutputStream, which can compress data in *xz* format. This API is based on open source *LZMA* SDK and is a part of *XZ Utils* project [2]. It uses the *LZMA* compression algorithm. Compressing data before storage can help in efficiently utilizing available storage space and before transmission can save network bandwidth. Table 4.4 shows the details of all the above Java compression APIs, MCOM1 (*zip*), MCOM2 (*gzip*) and MCOM3 (*xz*) along with their respective decompression APIs, MDECOM1, MDECOM2 and MDECOM3. Table 4.4 also contains the usage of these APIs by means of Java methods in the fifth column. *Buffer size* is a parameter which can be specified by a developer while using these APIs for compression and decompression.

Table 4.3: Various Compression APIs in Java

Format	API(s)	Notation	API Usage in a Program
zip	ZipOutputStream	MCOM1	1 OutputStream os = new FileOutputStream(compFile); 2 OutputStream zos = new ZIPOutputStream(os); 3 InputStream fis = new FileInputStream(srcFile); 4 byte **buffer**[] = new byte[**bufferSize**]; 5 int bytes_read; 6 while((bytes_read = fis.read(data,0,**bufferSize**))!=−1) 7 { 8 zos.write(**buffer**, 0, bytes_read); } 9 fis.close(); 10 zos.close();
gzip	GZipOutputStream	MCOM2	1 OutputStream os = new FileOutputStream(compFile); 2 OutputStream gos = new GZIPOutputStream(os); 3 InputStream fis = new FileInputStream(srcFile); 4 byte[] **buffer** = new byte[**bufferSize**]; 5 int bytes_read; 6 while((bytes_read = fis.read(**buffer**))>0) { 7 gos.write(**buffer**, 0, bytes_read); } 8 fis.close(); 9 gos.finish(); 10 gis.close();
xz	org.tukaani.xz .XZOutputStream	MCOM3	1 InputStream fis = new FileInputStream(srcFile); 2 OutputStream fos = new FileOutputStream(compFile); 3 InputStream bis = new BufferedInputStream(fis); 4 LZMA2Options opts = new LZMA2Options(); 5 opts.setPreset(5); 6 XZOutputStream xzos = 7 new XZOutputStream(fos,opts); 8 byte[] **buffer** = new byte[**bufferSize**]; 9 int bytesRead; 10 while((bytesRead = bis.read (**buffer**))!=−1) { 11 xzos.write (**buffer**, 0, bytesRead); } 12 xzos.close();

Table 4.4: Various Decompression APIs in Java

Format	API(s)	Notation	API Usage in a Program
zip	ZipInputStream	MDECOM1	1 ZipFile zipfile = new ZipFile(zipfile); 2 ZipEntry entry = ZipEntry.nextElement(); 3 InputStream is = new BufferedInputStream 4 (zipfile.getInputStream(entry)); 5 OutputStream fos = new 6 FileOutputStream(entry.getName()); 7 OutputStream bos = new BufferedOutputStream(fos); 8 byte[] **buffer** = new byte[**bufferSize**]; 9 int count; 10 while ((count = is.read(**buffer**)) != −1) { 11 bos.write(**buffer**, 0, count); } 12 bos.close(); 13 is.close();
gzip	GZipInputStream	MDECOM2	1 InputStream fis = new FileInputStream(gzipFile); 2 GZIPInputStream gis = new GZIPInputStream(fis); 3 OutputStream fos = new FileOutputStream(newFile); 4 byte[] **buffer** = new byte[**bufferSize**]; 5 int len; 6 while((len = gis.read(**buffer**)) != −1) 7 { 8 fos.write(**buffer**, 0, len); 9 } 10 fos.close(); 11 gis.close();
xz	org.tukaani.xz .XZInputStream	MDECOM3	1 InputStream fis = new FileInputStream(xzFile); 2 OutputStream fos = new FileOutputStream(output); 3 InputStream bis = new BufferedInputStream(fis); 4 XZInputStream xzIn = new XZInputStream(bis); 5 OutputStream bos = new BufferedOutputStream(fos); 6 byte[] **buffer** = new byte[**bufferSize**]; 7 int bytesRead; 8 while((bytesRead = xzIn.read(**buffer**)) != −1) { 9 bos.write (decoded, 0, bytesRead) ; } 10 xzIn.close();

Table 4.5: Server Machine Configuration

Type	Dell PowerEdge 2950
Processor	7x Intel Xeon, 3 GHz, 4 cores per processor
Hard Disk	1.7 Tera Bytes SAS
Main Memory	32 GB DIMM
Operating System	Linux(Ubuntu 14.04 LTS) Windows Server 2008
Java	JDK1.7.0

4.2 Experiments and Results

In this section, we show how developer's choice of APIs along with choosing an appropriate buffer size in performing file reading, file copy and file compression operations can impact the energy cost of the server. First, the energy cost of M1, M2, and M3 is measured with varying buffer sizes for reading a large file. Next, we evaluate the energy performance of MC1 and MC2 with different buffer sizes by copying a large file from one location to another. Next, we measure the energy cost of file copy by MC3, MC4, and MC5, and compare their energy costs with the the minimum and maximum energy costs of MC1 and MC2 at particular buffer sizes. Each buffer size in all the experiments is kept as a power of 2. The reason behind this as the most file systems are configured to use disk block sizes of 4096 or 8192. For example if buffer is configured to read 4100 bytes at a time, each read would require 2 block reads by the file system. Just few bytes (4) more than block size will be read in another read operation leads to inefficiency. That is why, most buffers sized as a power of 2 and generally larger than (or equal to) the disk block size. Then, the energy costs of MCOM1, MCOM2, and MCOM3 are compared by compressing a large pdf document. Finally, we measure the energy costs of decompression methods, MDECOM1, MDECOM2, and MDECOM3. All the experiments are performed on a Windows Server and a Linux Server. Table 4.5 shows the configuration of a rack mountable server along with the operating system details used to carry out the experiments.

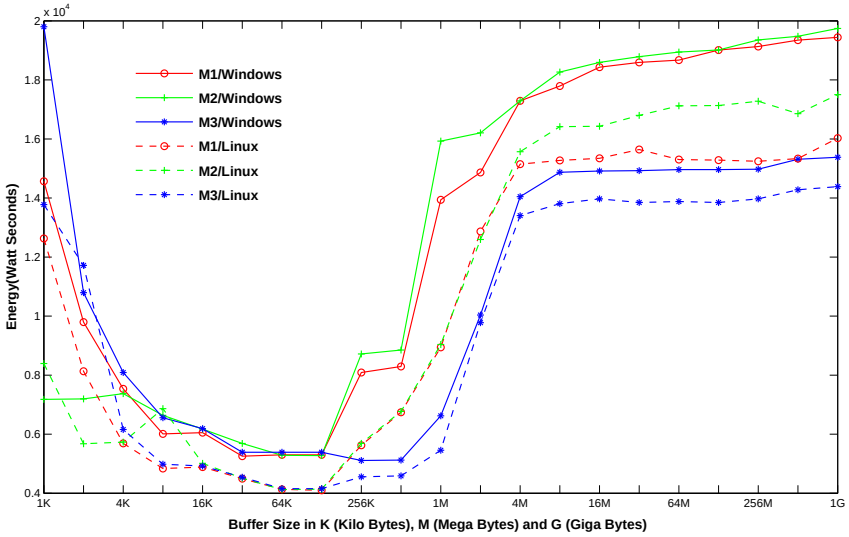

Figure 4.1: Energy cost of File Reading Methods M1, M2 and M3 with different buffer sizes

4.2.1 File Reading

We measure the total AC energy cost of a server for reading a 20 GB video file from a disk by using M1, M2 and M3 with different buffer sizes. A buffer size has been varied from 1KB (Kilo Byte) to 1 GB (Giga Byte). For every buffer size, each method is run 5 times and the average is taken to represent the energy cost for that particular buffer size. Figure 4.1 shows the energy cost of all the methods versus buffer size on both the platforms, Linux and Windows. It is clear from the graph, that all the methods follow the same trend in energy cost for the different buffer sizes on both the platforms. The energy cost started decreasing significantly from buffer size 1KB to 64KB, because with the increase in buffer size, a file will be read in big chunks thereby reducing the number of disk accesses. And in the range 1KB–32KB, M3 consumes more energy than M1 followed by M2. However, there is a sharp rise in the energy cost from 256KB to 4MB minimizing the effect of increased buffer size on energy cost. Then it remains constant till the buffer size is increased to 1GB. In the range 256KB–1GB, M2 consumes much more energy than M1 and M3. As the figure shows, at 64KB, all the methods consume almost equal and minimum energy. This energy cost patterns of all the File access methods are the same on both the operating systems. The reason behind the sharp rise in energy cost after 128KB is that the data from the disk is first copied into RAM, then into L2 cache, next into L1

cache and finally from L1, data is read by the CPU. For the large buffer sizes, the data of the buffer from one read call does not fit into L1 cache. Therefore it takes extra calls to transfer complete data from L2 to L1 cache in one read operation, increasing the waiting time between the consecutive read calls. However, the small buffer sizes fit into L1 cache; decreasing the latency between L2 and L1 caches.

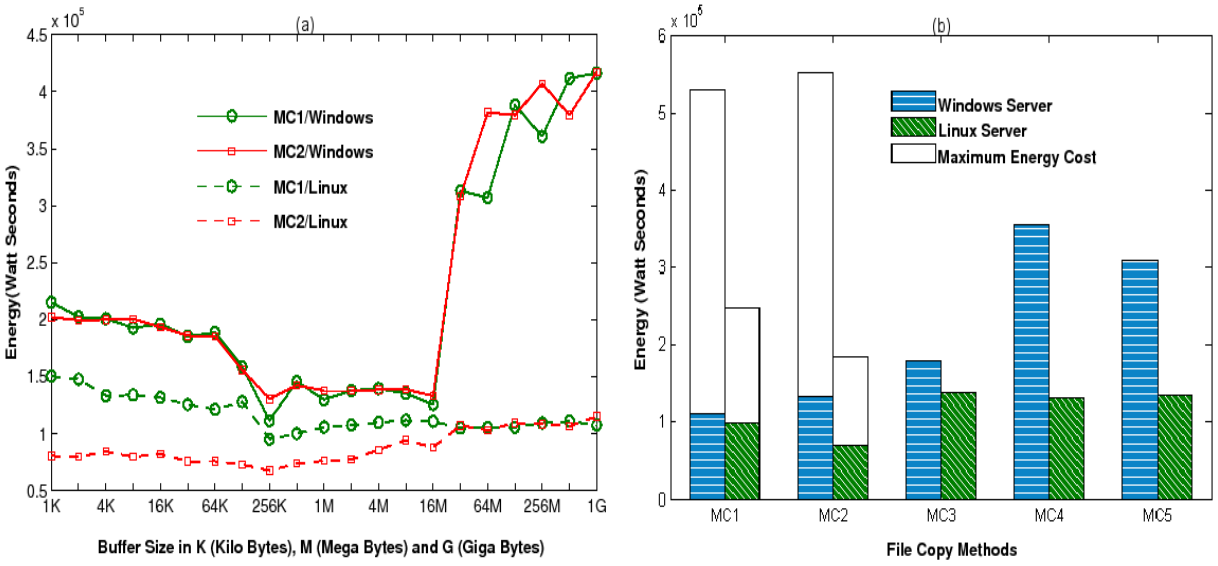

Figure 4.2: (a) Energy cost of file copy methods MC1 and MC2 with different buffer sizes on both Windows and Linux; (b) Comparsion of maximum and minimum energy cost of MC1 and MC2 at 256KB and 1GB buffer sizes, respectively, with the energy cost of MC3, MC4 and MC5.

4.2.2 File Copy

Figure 4.2(a) shows the energy performance evaluation of file copy methods, MC1 and MC2 at buffer size ranging from 1KB to 1GB. Energy cost is measured to copy a 20GB video file. The energy cost behaviour of MC1 and MC2 is similar to file reading methods, M1 and M2, respectively, as both are based on similar APIs (FileInputStream and BufferedInputStream, respectively). However, the minimum energy cost of MC1 and MC2 is at 256KB buffer size on both the operating systems. Next, we use the methods MC3, MC4 and MC5 to copy the same video file and measure the energy cost. A developer does not have any control over these methods as they do not have any tunable parameter which can be changed.

Figure 4.2(b) shows a bar graph which compares the energy cost of MC3, MC4 and MC5 with the minimum and maximum energy costs of MC1 and MC2. It is clear from the results that although it is easy for developers to use MC3, MC4 and MC5 to perform file copy operation, they consumes more energy than MC1 and MC2 with optimal buffer size, 256KB. After reading the source code of MC3 and MC5, it was found that these methods internally use buffers of fixed size of 2MB and 8MB, respectively, while copying files. This is the reason behind their more energy costs than the minimum energy cost of MC1 and MC2 at 256 KB. Developers might choose the APIs which are simpler to use without caring about their energy efficiency.

4.2.3 File Compression and Decompression

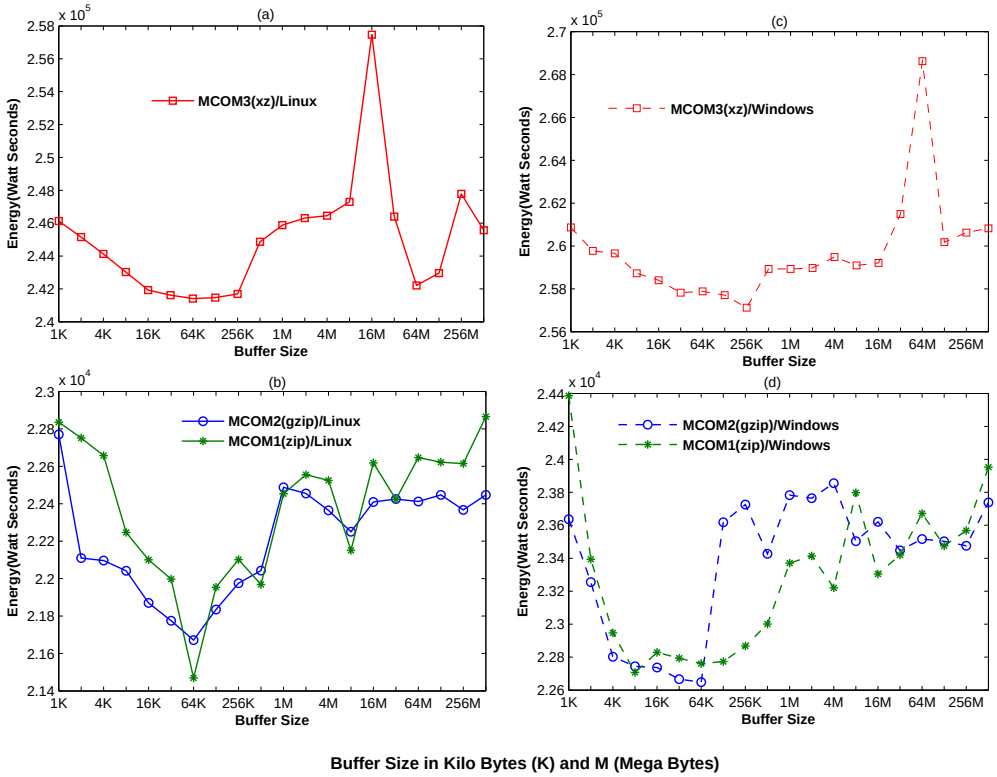

Figure 4.3: Energy cost of Compression APIs MCOM1, MCOM2 and MCOM3 with different buffer sizes on both Windows and Linux

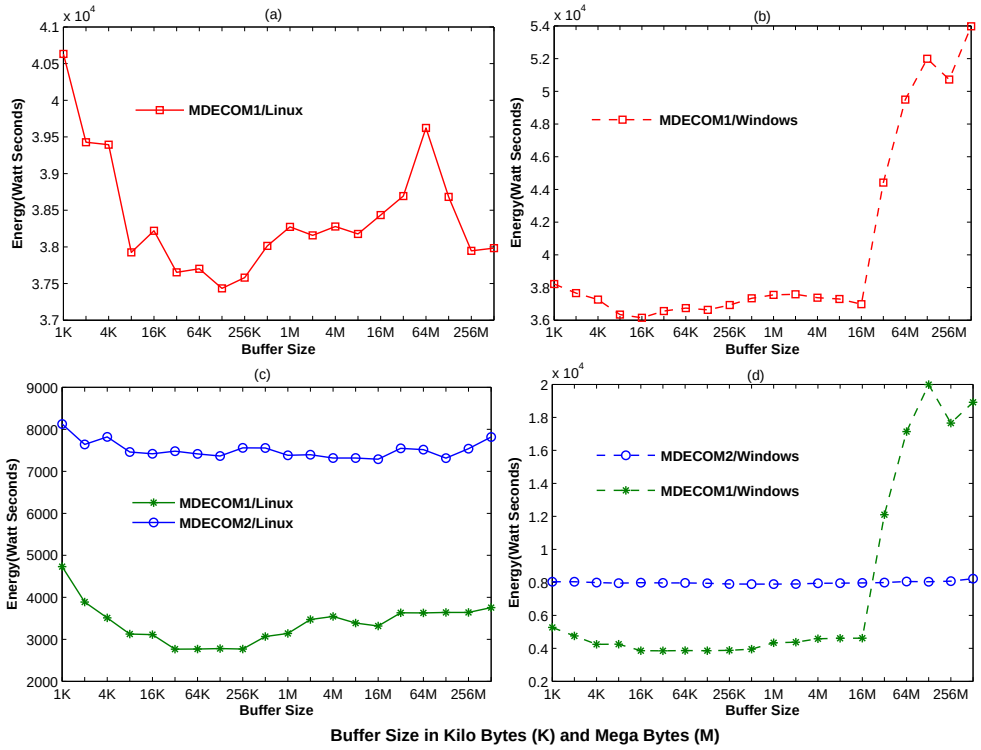

Figure 4.4: Energy cost of Decompression APIs MDECOM1, MDECOM2 and MDECOM3 with different buffer sizes on both Windows and Linux

The results of the measurement of energy cost of compressing a 1.2 GB pdf document by using MCOM1, MCOM2 and MCOM3 with different buffer sizes on both Windows and Linux servers has been shown in Figure 4.3. The change in the energy cost of MCOM1 and MCOM2 with varying buffer size is similar to that of file reading and file copy methods. However, for MCOM3 the energy trend is different; after 16 MB buffer size there is sharp rise in energy, after 64 MB, the energy started decreasing, and at 512 MB the energy cost becomes equal to the energy cost at 1KB buffer size. In addition, the energy cost is maximum for MCOM3. Energy consumption mainly depends on the effectiveness of compression, which is typically measured by Compression Ratio (CR) as defined below:

$$CR = \frac{Originalfilesize}{Compressedfilesize}$$

Table 4.6 shows the compression ratios of all the three compression APIs. MCOM1 and

MCOM2 have the same compression ratio. In other words, the size of the compressed files

Table 4.6: Compression Ratio of Different Methods

Method	**CR**
MCOM1	1.22
MCOM2	1.22
MCOM3	1.47

produced by these methods are the same. That is why there is not much difference in their energy costs. In contrast, MCOM3 has a better compression ratio, which is the reason for more energy cost. Further, it is clear from Figure 4.3 that choosing 64K buffer size for all the methods MCOM1, MCOM2 and MCOM3 leads to minimum energy compression on both the platforms. Similarly, Figure 4.4 shows the energy measurement results for decompression methods MDECOM1, MDECOM2 and MDECOM3 for decompressing the *zip*, *gzip* and *xz* files, respectively with, different buffer sizes. The graphs of the Figure 4.4 clearly show that decompressing a zipped file (MDECOM1) is most energy efficient on both the platforms, and MDECOM3 has maximum energy cost among all the methods. Even though MCOM1 and MCOM2 has almost the same energy cost, their respective decompression methods MDECOM1 and MDECOM2 have a large energy difference.

Then we calculate the percentage variation in the energy costs of all the methods with respect to buffer size by using the expression defined as below:

$$variation = \frac{E_{max} - E_{min}}{E_{max}} * 100$$

where E_{max} and E_{min} are the maximum and minimum energy costs of a certain file reading method, respectively, at particular buffer sizes. Table 4.7 shows the calculated variation in the energy costs of all the file reading, file copy, file compression and file decompression methods on both the platforms. The variation in the energy cost of file reading method, M2 is more as compare to methods, M1 and M3. For file copy, the energy cost of MC1 is more sensitive to the buffer size selection than of MC2. Next, from all the compression methods, the variation in energy cost with respect to buffer size is more for MCOM1 than for MCOM2 and MCOM3. Similarly, decompression by MDECOM1 is more sensitive to the choice of buffer size. It is evident from the Table 4.7, that developers need to be careful in selecting the buffer size while using methods, having high variation in energy costs with respect to buffer size.

Table 4.7: Perecntage variation in the energy costs of file reading methods

Method	%variation Windows Server	%variation Linux Server
M1	73%	74.2%
M2	73.2%	76.4%
M3	73%	69.8%
MC1	73.2%	37%
MC2	69%	41.33%
MCOM1	6.6%	6.1%
MCOM2	4.5%	3.4%
MCOM3	4.2%	6.23%
MDECOM1	79.6%	41.5%
MDECOM2	3.9%	8.7%
MDECOM3	33%	7.8%

4.3 Summary

We close this chapter by noting that in the real world, these operations (file reading, file copy, data compression and decompression) are being used million times on the data centers and choosing the right API with appropiate buffer size for a particular operation can lead to energy savings.

Chapter 5

Framework for estimating the energy cost of software applications on servers for various developer choices

5.1 Introduction

Large scale software applications running on servers need to be designed with power efficiency in mind. Although there are a number of ways to optimize the application at its design stage, as we described in Chapter 4; developers generally do not consider the energy cost of their software while making important design decisions. They find it difficult to measure the energy cost incurred by their workload and know how it behaves on real servers inside data centers. In addition to this, the measurement process takes a lot of human effort and time. As a result, modern data centers hosting big applications like Facebook, Gmail, Google, twitter etc. consume a major portion of world's electrical energy. Profiling the energy consumed in executing a particular task is essential to help software developers to build an energy efficient software applications. The measurement of power consumption of a server in data center is not good enough. It is important to know how much power a single application uses. Energy efficient application development needs information about the energy costs of hardware subsystems at the application level [20][26] and this can be accomplished by developing hardware and application profiles. Hardware profile tells about the energy consumption of hardware components by means of mathematical models, in contrast, for a given application, application profile provides information about the use of hardware subsystems. In real world, software applications are developed

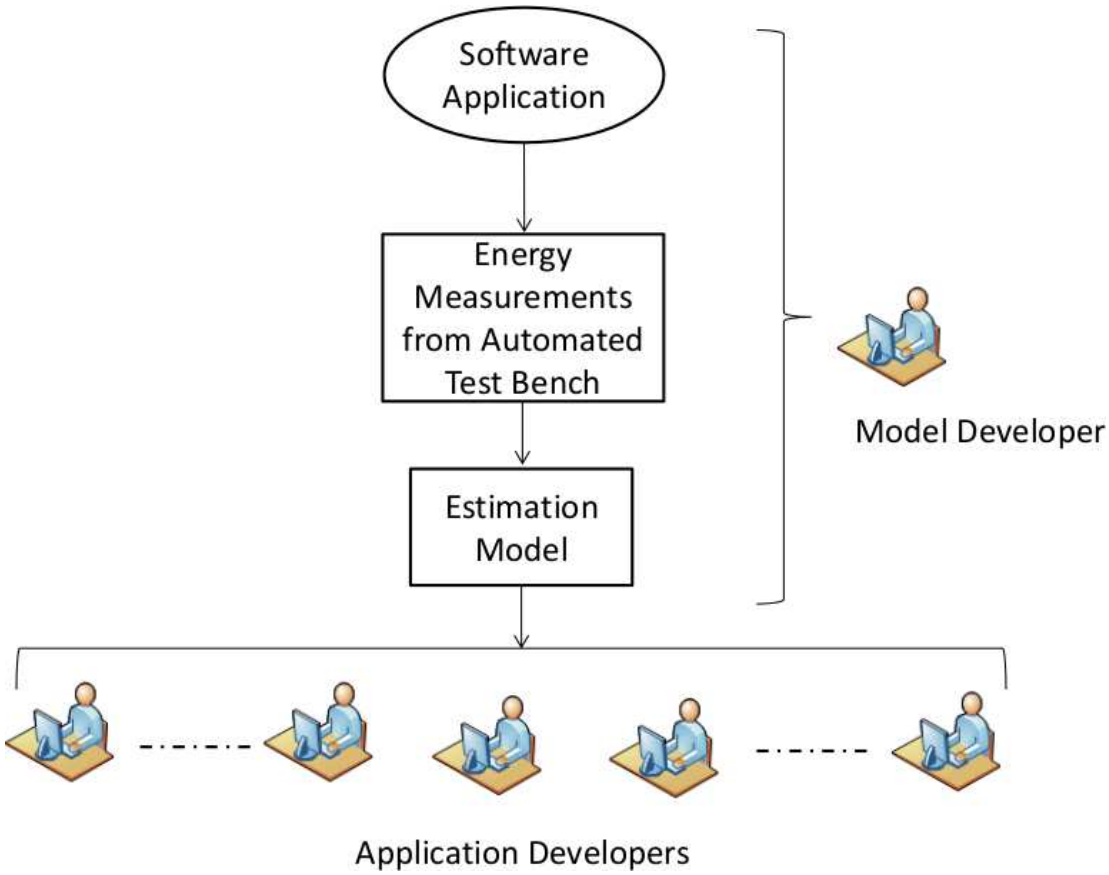

Figure 5.1: Framework

by a team of developers. For the energy efficient development, all the developers in a team need to choose energy efficient design options. If all the developers measure the energy for every design choice, it takes lot of human effort and time. To solve this problem, we present a framework in which for the common design options, only one developer measures the energy cost on a real server and develop energy cost models from the measurements. Other developers do need to measure the energy for the same options, they can use the models to estimate the energy consumption for those options.

5.2 Framework Description

1. **Software Application**: An application of which the developer is interested in measuring the energy cost on a real server for the various design choices. After estimating the energy cost, among all the choices, an energy efficient choice can be selected in final implementation of application.

2. **Energy Measurements from Automated Test Bench**: The main step in framework is to measure the actual energy cost of the application for the various design choices. For this, our automated test bench presented in Chapter 3: Figure 3.1 can be used. Among a team of developers, one developer known as Model developer can be assigned a task of preparing energy profiles of different design choices i.e taking energy measurements of all the design choices.

3. **Estimation Models**: When all the energy measurements are done, Model developer then prepares a mathematical models which estimate the energy cost of the applications for the various design choices. These models can then be used by application developers to predict the energy consumption for the same design option instead of taking direct measurements.

5.3 Example of using Framework in estimating the energy cost of File Reading Methods with different buffer sizes

In this section, we explain the usage of our framework in estimating the energy costs of all the file reading methods described in Chapter 4 with tunable parameter buffer size. We are interested in finding the energy costs of all the three file reading methods with different buffer sizes ranging from 1 KB to 1 GB. Once a model developer prepares mathematical models of the energy costs of reading methods with different buffer sizes using actual measurements from test bench, application developers can use those models to find the energy costs without taking actual measurements. In this way, application developers can decide on the buffer size for which the energy cost of file reading method is minimum. By using framework, the actual measurements are just needed once to build mathematical models after that in future, other developers deciding on the same design decisions do not need to spend time on measurements. They use the models to estimate the energy costs.

5.3.1 Energy measurements from test bench

The first step in the framework, is to measure the real energy costs from test bench. We take the example of file reading method FileInputStream (M1) described in Chapter 4 to demonstrate the usefulness of framework. First the energy cost of M1 is measured with different buffer sizes ranging from 1K–512MB using our automated test bench. Figure 5.2 shows the energy measurement results of the API in the form of graph.

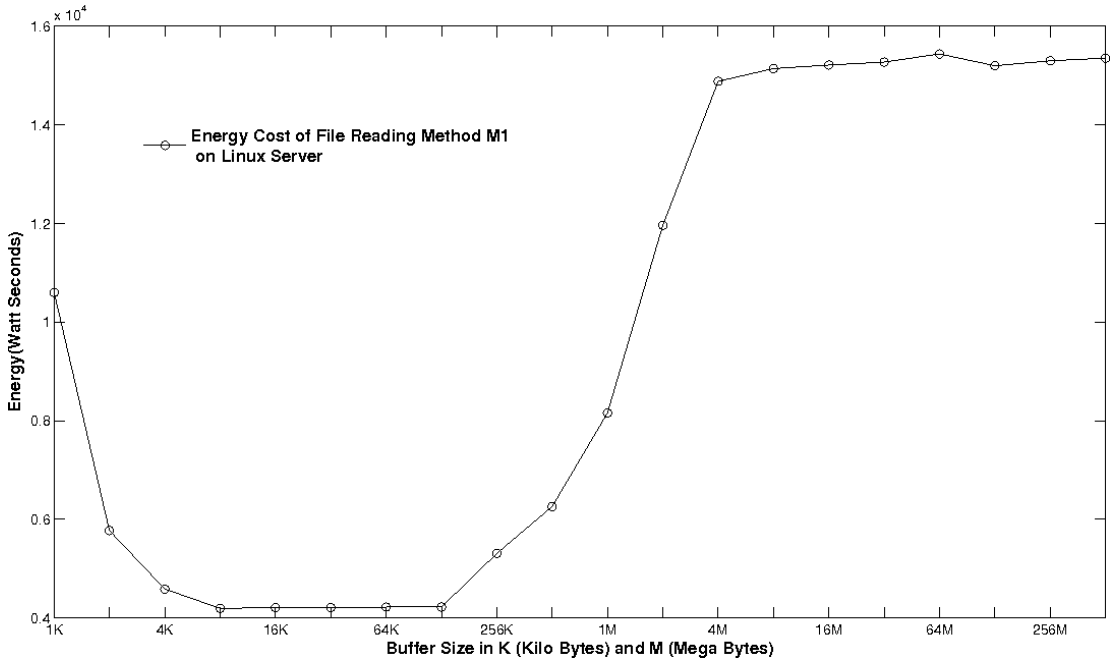

Figure 5.2: Energy cost of File Reading Method M1 with different buffer sizes on Linux Server

5.3.2 Modelling energy costs from measurements

In this section, we estimate the energy costs of APIs with different buffer sizes on a particular platform by means of regression modelling. We model the energy costs using two techniques: Polynomial regression and Splines which are discussed below:

1. **Polynomial Regression** : Polynomial regression is a form of multiple regression technique in which the relationship between the independent variable x and the

dependent variable y can be modelled as an n^{th} degree polynomial. For x and y, the polynomial equation can be written as:

$$y = C_0 + C_1x + C_2x^2 + C_3x^3 + C_4x^4 + \cdots + C_nx^n,$$

where n is the degree of the polynomial and C_i, $0 \leq i \leq n$, is a constant In our case, the independent variable is buffer size and dependent variable is energy cost. Therefore, in terms of buffer size (b), the energy cost can be modelled as:

$$EnergyCost_b = C_0 + C_1b + C_2b^2 + C_3b^3 + C_4b^4 + \cdots + C_nb^n.$$

We use the Matlab function $polyfit$ to compute the constants in the polynomial equation.
Function $polyval(p, x)$ returns the value (energy cost) of a polynomial (p) of degree n evaluated at x (buffer size).

Assessing Goodness of Fit: The quality of the fit requires assessing of goodness of fit. It involves a least-squares approximation; the distance of the entire set of data points from the fitted curve. The normalization of the residual error minimizing the square of the sum of squares of all residual errors. The norm of the residuals indicates a better fit as its value approaches zero.

Listing 5.1 contains the Matlab code for modelling the experimental data using polynomial regression. The residual norm and indicate goodness of fit. Line# 1 and 2 generates the polynomial coefficients for the experimental data for given degree N. Lines #3, 4 and 5 calculate the variation of predicted data from the original data. Line# 7 shows code for calculation of norm. The closer that norm is to 0, the more completely the fitted model explains the data.

Listing 5.1: Matlab code for modelling the data by Polynomial Regression

```
1 PN = polyfit(x,y,N);
2 ypred = polyval(PN,x);      %Predicted Data
3 dev = y - mean(y);          %deviations - measure of spread
4 SST = sum(dev.^2);          %total variation to be accounted for
5 resid = y - ypred;          %residuals - measure of mismatch
6 SSE = sum(resid.^2);        %variation NOT accounted for
7 normr = sqrt(SSE);          %the 2-norm of the vector of the residuals for the fit
```

To apply the polynomial regression to the energy cost data, we try with the different degrees of polynomials and we finalize the model whose norm is minimum. Figures

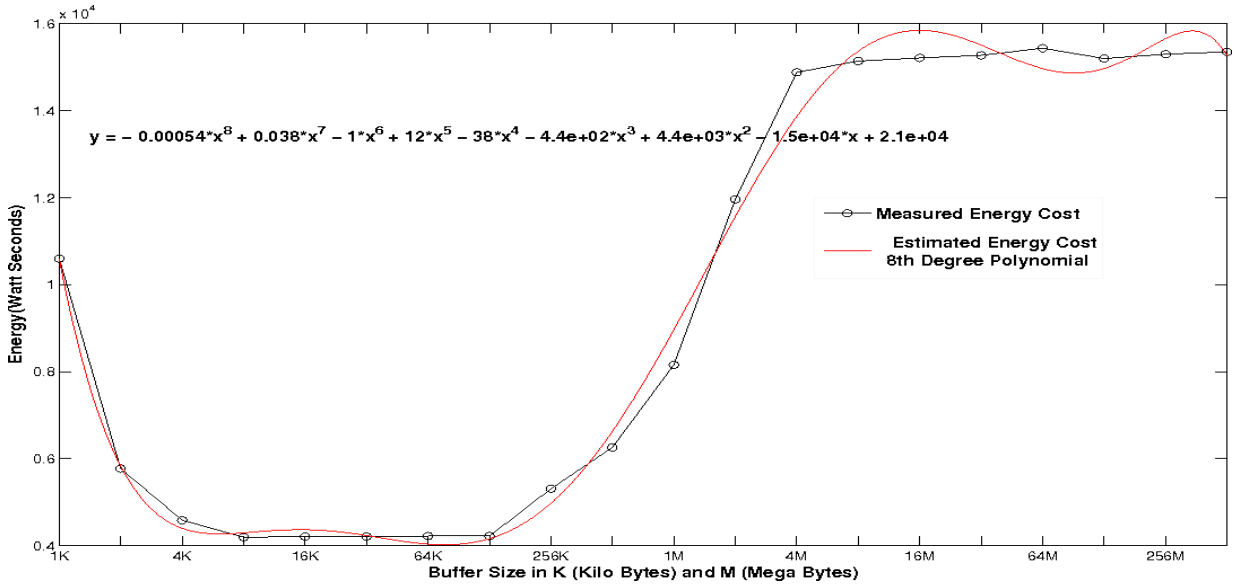

Figure 5.3: Energy Cost Estimation of M1 (File Reading Method) on Linux Server by Polynomial of degree 8 with norm=1767.6

5.3, 5.4, 5.5 shows measured energy costs as well as predicted energy costs with polynomials of degrees 8, 9 and 10 respectively for M1(FileInputStream) method on Linux platform. Calculated norm for the three degrees is also shown is captions of figures. The norm for tenth degree polynomial is minimum, therefore its coefficients make the energy profile for M1 method on Linux Server which can be written as:

Energy Profile = (3.3261e-05 -0.00341 0.14943 -3.6605 55.15 -531.1 3305.9 -13166 32539 -46468 34870)

2. **Spline** : Sometimes a single polynomial is not good enough to model a data, therefore a solution is to use several polynomials pieced together. A spline is a numeric function that is piecewise-defined by polynomial functions, and which possesses a sufficiently high degree of smoothness at the places where the polynomial pieces connect; known as knots [1]. Cubic spline interpolation is made of different cubic polynomials. "spline" function of curve fitting toolbox is used model the energy cost of M1 for different buffer sizes . Figure 5.6 shows the estimated energy cost along with measured cost of M1 method by spline interpolation.

   Spline interpolation gives accurate estimation of energy cost as the norm of residuals

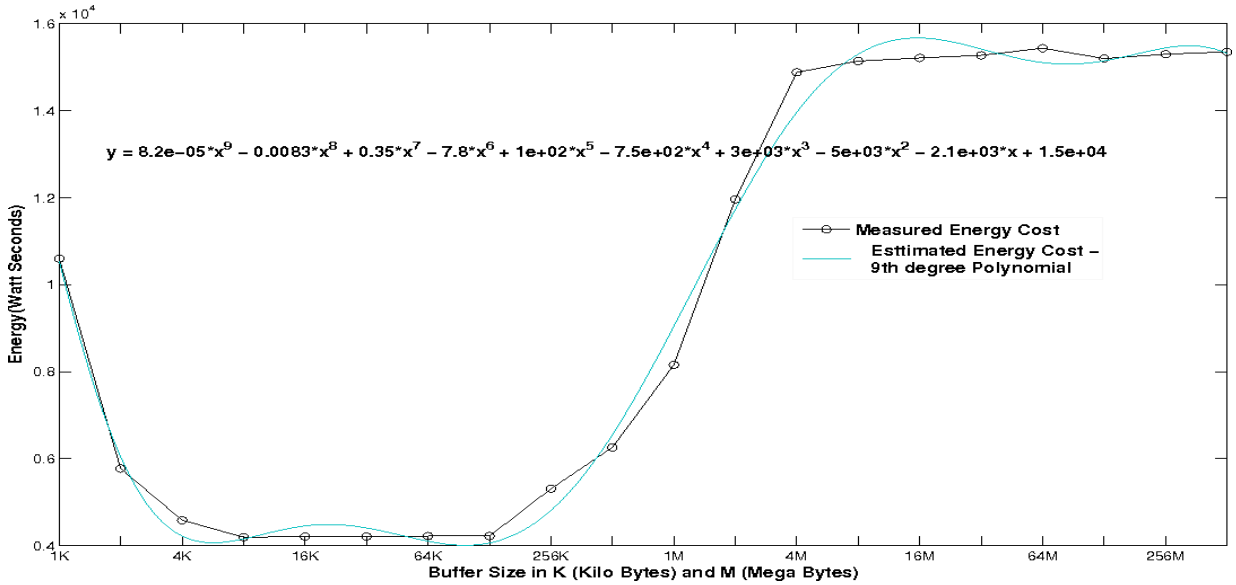

Figure 5.4: Energy Cost Estimation of M1 (File Reading Method) on Linux Server by Polynomial of degree 9 with norm=1660.6

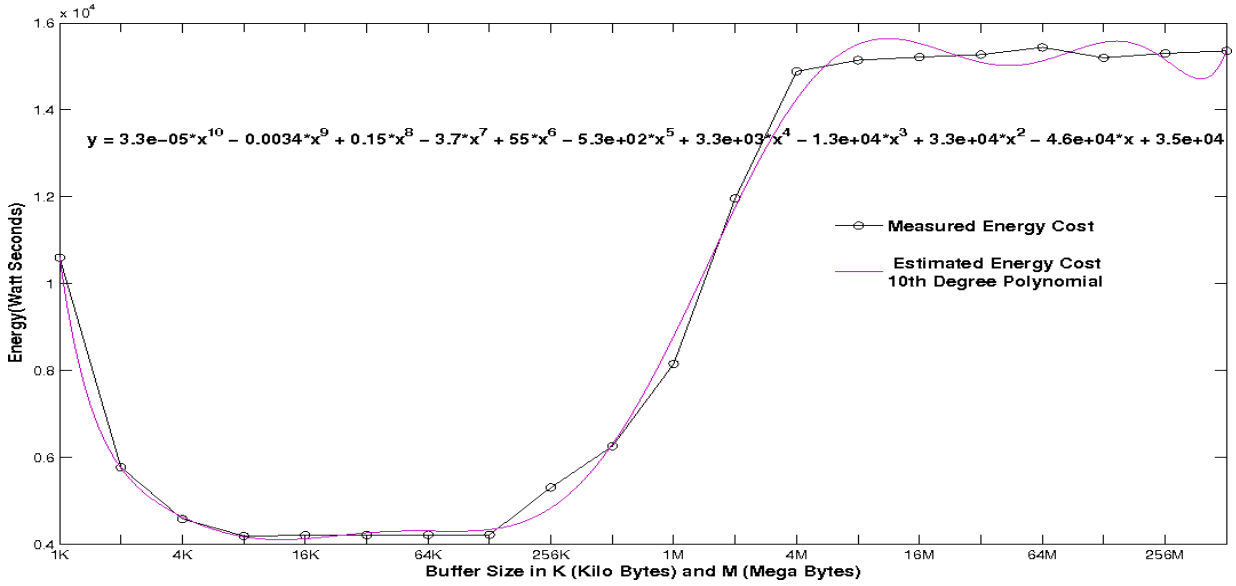

Figure 5.5: Energy Cost Estimation of M1 (File Reading Method) on Linux Server by Polynomial of degree 10 with norm=1276.5

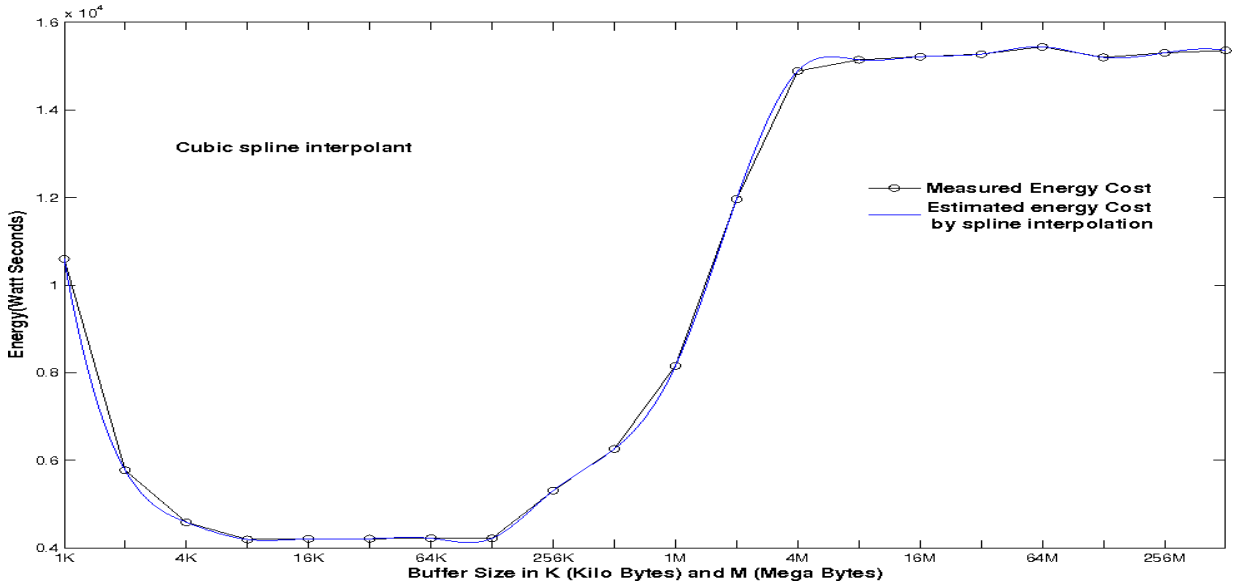

Figure 5.6: Energy Cost Estimation of M1 (File Reading Method) on Linux Server by Spline interpolation with norm=0

is 0 than the polynomial regression modelling.

5.3.3 Making models available to the other developers

Once the model for energy costs for File Reading method has been prepared by model developer then that model will be available to application developers. While deciding on the buffer size to use for M1, developers estimate the energy cost by using model instead of wasting time on direct measurements.

5.4 Summary

In this Chapter, we presented a design of a framework in which one developer generates energy cost models for the common design options. Afterwords, other developers can make use of those models to find the energy costs for the same design options instead of direct measurements.

Chapter 6

Conclusion and Future Work

The design of a software application has a significant impact on the power consumption. With the aim of reducing power bills of data centers, "Green Computing" has emerged with the primary goal of making software more energy efficient without compromising the performance. Developers play an important role in controlling the energy cost of data center software while writing code. In Chapter 2, we provide a comprehensive literature review of energy measurement approaches and compared our approach with them. In addition, various techniques to save energy costs of data centers and existing research on the energy efficiency of software applications are discussed.

In Chapter 3,we presented an automation framework to measure the energy cost of servers while running software applications. The framework's infrastructure mainly contains a power meter, target server and control software (PAST) for synchronization and monitoring. By using the test bench, we performed actual measurements to verify the claim in a previously published paper [5] that energy cost of reading files by the method FileInputStream (M1) is greater than the BufferedInputStream (M2) method. However this claim is not valid in certain cases, if we introduce a programmer buffer in both the methods. It holds good for buffer sizes ranging from 128 bytes till 8KB, but these two methods consume almost the same energy at buffer sizes from 8KB to 64MB. Also, the introduction of *buffer* in M2 has further reduced its energy cost. Finally, we compared the energy costs of the same functionality provided by different software applications by measuring the energy costs of compression and decompression features of two Linux packages: *7z* and *rar*. The *7z* package consumes more energy than rar in compressing and decompressing files. However, *rar* consumes more energy in compressing to *.7z* format than to *.rar* format. The automation framework can be used by programmers to evaluate the energy cost of their applications while making important design decisions.

In Chapter 4, we showed by means of experiments that developers can make coding decisions to reduce the energy cost of software by choosing energy efficient APIs with optimal choice of parameters for performing a particular operation. We performed actual power measurements of various APIs available in Java to carry out four categories of operations, file reading, file copy, file compression and decompression, on a real server Dell Power Edge 2950, using our automated test bench. These operations are prevalent in large-scale data-intensive computing applications running in modern data centers. For all the methods, which have buffer size as tunable parameter, energy cost is measured with different buffer sizes. Our results show that:

- The choice of a particular API for file reading only matters for the *buffer size* ranges, 1KB–32KB and 128KB–1GB because at 64KB *buffer size*, all the file reading APIs, FileInputStream, BufferedInputStream and FileChannel, consume minimum energy. In addition, the energy costs of all the methods are sensitive to the *buffer size* selection because the variation in their energy costs lie in the range 73%–76.4% with respect to *buffer size*
- A developer choice of BufferedOutputStream (MC2) at 256 KB *buffer size* for file copy is more energy efficient than FileOutputStream (MC1) and methods using fixed size *buffers*, namely, Java 7 Files.Copy, Apache FileUtils.copyFile, and FileChannel transfer method.
- The compression ratio of *xz* compression is larger than the compression ratios of *zip* and *gzip*, thereby consuming more energy. Similarly, decompression from *xz* file is least energy efficient. However, the energy costs of *zip* compression and its decompression are more sensitive to a particular choice of *buffer size*.

In Chapter 5, we presented a design of a framework in which one developer generates energy cost models for the common design options. Afterwords, other developers can make use of those models to find the energy costs for the same design options instead of direct measurements.

6.1 Limitations and Future Work

In this section, we provide some of the limitations of the this thesis. In addition, we also highlight a scope for future work to overcome these limitations. Following are the points which discuss about the limitations and future work.

- We presented an automated test bench whereby developers can measure the energy costs of their applications on a server during the coding stage of software life-cycle. At present, the monitoring station has to be physically connected to the meter and test server. Also, developer has to be at the same location where server and meter resides. In future, this limitation can eliminated by making the monitoring station to collect the readings from the meter and control the server over the network. Therefore, from any location developers would measure the energy cost of their code by controlling the server and meter over the network.

- More work is required to validate the effects of cache size, block size and memory page size on the energy costs of APIs with different buffer sizes. We can perform the experiments on the different machines with different cache sizes.

- Our proposed framework for estimating the energy costs of APIs can integrated with our automated test bench to offer it as an online service to developers. Therefore, developers would be able to evaluate their code for power consumption.

- More work is required to prepare energy efficient guidelines which programmers can refer while developing applications.

References

[1] Spline(mathematics). *http://www.wikipedia.org*.

[2] Xz utils. *http://tukaani.org/xz/*, 2011.

[3] A. Abogharaf and K. Naik. Client-centric data streaming on smartphones: An energy perspective. In *Mobile and Wireless Networking (MoWNet)*, 2013.

[4] Abdullah Al Hasib and Sanjeeda Sharmin. Green software: Recent trends and techniques for software design and development. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, 2(1):pp–077, 2013.

[5] Luca Ardito, Giuseppe Procaccianti, Antonio Vetro, and Maurizio Morisio. Introducing energy efficiency into sqale. In *ENERGY 2013, The Third Intl. Conf. on Smart Grids, Green Communications and IT Energy-aware Technologies*, pages 28–33.

[6] Sergio Barrachina, Maria Barreda, Sandra Catalán, Manuel F Dolz, Germán Fabregat, Rafael Mayo, and Enrique S Quintana-Ortí. An integrated framework for power-perf. analysis of parallel scientific workloads. In *ENERGY 2013, The Third Intl. Conf. on Smart Grids, Green Communications and IT Energy-aware Technologies*, pages 114–119.

[7] David J Brown and Charles Reams. Toward energy-efficient computing. *Communications of the ACM*, 53(3):50–58, 2010.

[8] Eugenio Capra, Giulia Formenti, Chiara Francalanci, and Stefano Gallazzi. The impact of mis software on it energy consumption. 2010.

[9] Eugenio Capra, Chiara Francalanci, and Sandra A Slaughter. Is software green? application development environments and energy efficiency in open source applications. *Information and Software Technology*, 54(1):60–71, 2012.

[10] Qiaozhen Chai, Zhongzhi Luan, Depei Qian, Ming Xie, and Wei Chen. Empowering designers to estimate function-level power for developing green applications. In *2013 International Conference onCloud and Service Computing (CSC)*, pages 57–62. IEEE, 2013.

[11] Zehan Cui, Yan Zhu, Yungang Bao, and Mingyu Chen. A fine-grained component-level power measurement method. In *International Conference on Green Computing and Workshops (IGCC)*, pages 1–6. IEEE, 2011.

[12] Jack W Davidson and Sanjay Jinturkar. Memory access coalescing: a technique for eliminating redundant memory accesses. In *ACM SIGPLAN Notices*, volume 29, pages 186–195. ACM, 1994.

[13] Carpe Diem. Java fast io using java.nio api. *http://www.idryman.org/blog/2013/09/28/java-fast-io-using-java-nio-api/*, 2011.

[14] Krisztina Erdelyi. Special factors of development of green software supporting eco sustainability. In *2013 IEEE 11th International Symposium on Intelligent Systems and Informatics (SISY)*, pages 337–340. IEEE, 2013.

[15] Miguel A Ferreira, Eric Hoekstra, Bo Merkus, Bram Visser, and Joost Visser. Seflab: A lab for measuring software energy footprints. In *2nd Intl. Workshop on Green and Sustainable Software(GREENS)*, pages 30–37. IEEE, 2013.

[16] Martin Fowler. Refactoring: Improving the design of existing code, 1997.

[17] Christopher W Fraser, David R Hanson, and Todd A Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(3):213–226, 1992.

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

[19] Hazem Hajj, Wassim El-Hajj, Mehiar Dabbagh, and Tawfik R Arabi. An algorithm-centric energy-aware design methodology.

[20] Shuai Hao, Ding Li, William GJ Halfond, and Ramesh Govindan. Estimating mobile application energy consumption using program analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 92–101. IEEE, 2013.

[21] Robert Harmon, Haluk Demirkan, Nora Auseklis, and Marisa Reinoso. From green computing to sustainable it: Developing a sustainable service orientation. In *2010 43rd Hawaii International Conference on System Sciences (HICSS)*, pages 1–10. IEEE, 2010.

[22] Chen-Wei Huang and Shiao-Li Tsao. Minimizing energy consumption of embedded systems via optimal code layout. *IEEE Transactions on Computers*, 61(8):1127–1139, 2012.

[23] Nattachart Ia-Manee and Peraphon Sophatsathit. Reducing engergy consumption in programs using cohesion technique. *International Journal of Computer Theory and Engineering*, 5(4):621–625, 2013.

[24] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 93. IEEE Computer Society, 2003.

[25] Jasmeet Singh, Kshirasagar Naik and Veluppillai Mahinthan. Automation of energy performance evaluation of software applications on servers. In *Proceedings of SERP'14: International Conference on Software Engineering, Research and Practice, Las Vegas Nevada, US*, page 7, 2014.

[26] Aman Kansal and Feng Zhao. Fine-grained energy profiling for power-aware application design. *ACM SIGMETRICS Performance Evaluation Review*, 36(2):26–31, 2008.

[27] J Koomey. My new study of data center electricity use in 2010. *Koomey. com*, 2011.

[28] Rachita Kothiyal, Vasily Tarasov, Priya Sehgal, and Erez Zadok. Energy and performance evaluation of lossless file data compression on server systems. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, page 4. ACM, 2009.

[29] John C McCullough, Yuvraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C Snoeren, and Rajesh K Gupta. Evaluating the effectiveness of model-based power characterization. In *USENIX Annual Technical Conf*, 2011.

[30] Dustin McIntire, Kei Ho, Bernie Yip, Amarjeet Singh, Winston Wu, and William J Kaiser. The low power energy aware processing (leap) embedded networked sensor system. In *Proceedings of the 5th intl. conf. on Information processing in sensor networks*, pages 449–457. ACM, 2006.

[31] Huzefa Mehta, Robert Michael Owens, Mary Jane Irwin, Rita Chen, and Debashree Ghosh. Techniques for low energy software. In *Proceedings of the 1997 international symposium on Low power electronics and design*, pages 72–75. ACM, 1997.

[32] Donald Melanson. My new study of data center electricity use in 2010. *http://www.engadget.com/2011/04/26/visualized-ring-around-the-world-of-data-center-power-usage/*, 2011.

[33] Trevor Mudge. Power: A first class design constraint for future architectures. In *High Perf. Computing*, pages 215–224. Springer, 2000.

[34] K. Naik. A survey of software based energy saving methodologies for handheld wireless communication devices. In *Dept. of Electrical and Computer Eng., Technical Report, TR-2010-13*.

[35] Kshirasagar Naik and David SL Wei. Software implementation strategies for power-conscious systems. *Mobile Networks and Apps*, 6(3):291–305, 2001.

[36] Adel Noureddine, Aurelien Bourdon, Romain Rouvoy, and Lionel Seinturier. A preliminary study of the impact of software engineering on greenit. In *2012 First International Workshop on Green and Sustainable Software (GREENS)*, pages 21–27. IEEE, 2012.

[37] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. A review of energy measurement approaches. *ACM SIGOPS O.S. Review*, 47(3):42–49, 2013.

[38] Jae-Jin Park, Jang-Eui Hong, and Sang-Ho Lee. Investigation for software power consumption of code refactoring techniques.

[39] Luigia Petre. Energy-aware middleware. In *15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2008, ECBS 2008*, pages 326–334. IEEE, 2008.

[40] Vijay Raghunathan, Cristiano L Pereira, Mani B Srivastava, and Rajesh K Gupta. Energy-aware wireless systems with adaptive power-fidelity tradeoffs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(2):211–225, 2005.

[41] Manuj Sabharwal, Abhishek Agrawal, and Grace Metri. Enabling green it through energy-aware software. *IT Professional*, pages 19–27, 2013.

[42] Cagri Sahin, Furkan Cayci, James Clause, Fouad Kiamilev, Lori Pollock, and Kristina Winbladh. Towards power reduction through improved software design. In *Energytech, IEEE*, pages 1–6, 2012.

[43] Daniel Schall, Volker Hudlet, and Theo Härder. Enhancing energy efficiency of database applications using ssds. In *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, pages 1–9. ACM, 2010.

[44] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. An energy consumption framework for distributed java-based systems. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 421–424. ACM, 2007.

[45] Jeffrey Shafer, Scott Rixner, and Alan L Cox. The hadoop distributed filesystem: Balancing portability and performance. In *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), 2010*, pages 122–133. IEEE, 2010.

[46] Vaidyanathan Srinivasan, Gautham R Shenoy, Srivatsa Vaddagiri, Dipankar Sarma, and Venkatesh Pallipadi. Energy-aware task and interrupt management in linux. In *Ottawa Linux Symposium*, 2008.

[47] Thanos Stathopoulos, D Mclntire, and William J Kaiser. The energy endoscope: Real-time detailed energy accounting for wireless sensor nodes. In *International Conference on Information Processing in Sensor Networks, 2008. IPSN'08*, pages 383–394. IEEE, 2008.

[48] B Steigerwald and Abhishek Agrawal. Developing green software. *Intel White Paper*, 2011.

[49] Yuwen Sun, Lucas Wanner, and Mani Srivastava. Low-cost estimation of sub-system power. In *Intl. Green Computing Conference (IGCC)*, pages 1–10. IEEE, 2012.

[50] Peter AH Peterson Digvijay Singh William, J Kaiser, and Peter L Reiher. Investigating energy and security trade-offs in the classroom with the atom leap testbed. 2011.

[51] Zichen Xu. Building a power-aware database management system. In *Proceedings of the Fourth SIGMOD PhD Workshop on Innovative Database Research*, pages 1–6. ACM, 2010.

[52] Heng Zeng, Carla S Ellis, Alvin R Lebeck, and Amin Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *ACM SIGPLAN Notices*, volume 37, pages 123–132. ACM, 2002.

[53] Chenlei Zhang, Abram Hindle, and Daniel M German. The impact of user choice on energy consumption. *Software, IEEE*, 31(3):69–75, 2014.

[54] Xiao Zhang, Jian-Jun Lu, Xiao Qin, and Xiao-Nan Zhao. A high-level energy consumption model for heterogeneous data centers. *Simulation Modelling Practice and Theory*, 39:41–55, 2013.