Politecnico di Milano
Faculty of Engineering
Master Degree in Computer Engineering

A SOFTWARE CACHING METHODOLOGY FOR POWER OPTIMIZATION IN DATA CENTERS

Tutor: Prof. Francalanci Chiara
Co-Tutors: Capra Eugenio
Giampaolo Agosta

Students:
Katsumi Fabio
Maica Reis Igor

Academic Year 2009-2010

*Agradecimento às nossas famílias,
pois sem base consistente não há
crescimento pessoal sólido.*

Abstract

Nowadays, the evidence of human environmental impact is easily noticed. Climatic changes and natural resources depletion have triggered the world concerns. The increasing costs of energy and its influence on companies budget regarding the IT infrastructure, i.e., Data Centers, have led enterprise focus to power optimization. This research aims to create a software enhancement to reduce the energy required by equipments since there are indications that software redounds consumption. The thesis presents a software caching methodology to reduce the computation time in mathematical high workload systems commonly used in financial institutions. Considering that fully stressed hardware demands more power, the time saved induces the energy reduction. Furthermore, a numerical model was engendered for software methods assortment and cache decisions. After the development, a set of tests was performed and the defined metrics and measurements turned out approximately 20% of energy savings. The work accomplished showed us up that a software layer optimization may produce powering reduction beside time computation decreasing. This implies the cutback of energy demand through all segments of a Data Center infrastructure.

Sommario

Oggi, l'evidenza dell'impatto ambientale umano è facilmente notata. I cambiamenti climatici e l'esaurimento delle risorsi naturali hanno innescato le preoccupazioni del mondo. I costi crescenti di energia e la sua influenza sul bilancio delle aziende per quanto riguarda l'infrastruttura IT, cioè Data Centers, hanno direzionato il fuoco per ottimizzazione della potenza. Questa ricerca mira la creazione di un miglioramento nel software per ridurre l'energia richiesta dagli apparecchi perchè ci sono indicazione che il software incide nel consumo energetico. La tesi presenta una metodologia di cache in software per abbassare il tempo di calcolo matematico in sistemi con alto carico di lavoro di uso comune nelle istituzioni finanziarie. Considerando che le hardware più usati richiedono più potenza, il tempo risparmiato induce la riduzione del consumo energetico. Inoltre, un modello numerico è stato generato per assortimento dei metodi del software e le decisioni della cache. Dopo lo sviluppo, una serie di test è stato effettuato ei parametri definiti e misurazioni ci è rivelato circa il 20 % di risparmio energetico. Il lavoro svolto ci ha mostrato che su una ottimizzazione nel livello del software può produrre riduzione della potenza insieme con il tempo di calcolo. Ciò implica la riduzione della domanda di energia attraverso tutti i segmenti di una infrastruttura di Data Center.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Currently many institutions and committees are being formed to upgrade the efficiency of power consumption. Power became a very important issue regarding the sustainable development and care of natural resources of our planet. Thus, organizations are facing the need to develop energy efficient data centers.

No doubts data centers are continuously growing fast and in high proportions, as energy costs increases in even higher rates. Consequently, energy has turned into a very important part of companies budget. This has led to the creation of a research field focused in power consumption and IT optimizations called Green IT.

The term green came up as an analogy to world's natural resources and the relation of its preservation. It is known that energy production is related to the consumption of our exiguous and finite environmental resources. Hence, energy efficiency has become important and has been analyzed deeply. It can be characterized in three distinct aspects.

The first concerns whole energy needed by a Data Center taking into account cooling systems, machines powering, overall infrastructure and its continuous increasing costs. The next aspect is the Data Center and its expansions. Usually, IT structures dwell in high density regions with limited infrastructure flexibility. So, the second aspect is related to scalability and the hardness for power providers to supply the necessary energy. In the third one can be considered the environmental impact caused by non-optimized resources being used in Information Technology sector.

Usually, the analysis of power consumption and optimization stops in direct causes like cooling systems but more in-depth researches are needed to identify higher-levels drivers. This case - we are going to verify in this work - proves the indirect participation of software executions in energy consumptiom. We mention Operational Systems, Drivers, firmware and ERP as important participants of behavior changing in IT equipments and therefore they are indirectly responsible of the overall consumption.

Nowadays, the software engineering still has the main focus bind to performances and classic software quality metrics where energy optimization and consumptions are not taken into consideration. This affects also the enterprise world where trade-off among costs and performances does not esteem software energy efficiency. If we specialize the software field, we can focus in the financial world, where computation and mathematical processes are heavy and demand a complete IT infrastructure (important budget).

This thesis refers a wider project which analyzes Green IT energy efficiency issues and which has been developed in collaboration with Politecnico di Milano and italian financial institutions. This document was structured as a brief introduction describing the scenario of Green Researches with motivations and objectives followed by some details regarding the Green Area and some important concepts. Then, will be presented the project itself with details about the analysis, development, execution and result interpretations.

Chapter 2

State Of The Art

This chapter will describe the Green IT and its current state of art. The following sections present a brief overview of this research field, motivations with statistical data and also a set of foundations which have been created to handle this environmental issue.

2.1 Green IT

Nowadays technology is an important part of our lives. The use of technology has exploded in several areas, improving life and work offering great advantages. Almost everything has a participation in technology and sometimes we do not even notice.

The expression Green IT is from now on commonly used to indicate a new field focused on problems related to environmental impact and the energy consumption of information systems. Particularly, the term can be referred to two specific themes: the IT energetic efficiency and the management of ecological compliancy of IT lifecycle. Both of them are directly related to the control of natural resources usage and the known impact of the required energy production.

More precisely, the first theme concerns the improvement of energetic efficiency by means of infrastructure management and systems changing avoiding any influence in already used systems and the way they are used. We would say: a kind of external layer of optimization on an atomic system. Its important to be mentioned that in last years the ratio between computational capacity and energy required has become better. This was mainly motivated by mobility and low-power devices.

The second theme looks upon the lifecycle of IT equipments development, energy needed in production, power consumption during the usage cycle and more.

2.1.1 Motivation

Statistical data related to Green Research and some facts are being published very often. Nothing better them practical data to justify the current growth of Green Field. Illustrating this, we see that in beginning of 21st century approximately 72% of whole energy used by a Personal Computer was spent in its production while the others 24% consumed in its 4 years mean usage. Since performance and high frequencies was for a long time and still motivate designers, power consumption increased considerably. Now, the statistics turn up that the percentages inverted - 23% of whole energy consumed by the PC (industry and usage) required in its production and 72% during 4 years of utilization. Then, we infer that production process has been optimized but the equipments with high performances and the current users profile requires a huge amount of energy.

4 Years Lifecycle					
Source	Procution	Dist	Use	End of Life	Total MJ
[1]	72%	4%	24%		8416
[2]	76%		23%	<1%	7633
[3]	23%	5%	72%	<1%	14264

Table 2.1: Energy Consumption During PC Lifecycle

It is important to mention that is also considered a slice of whole power consumption the disposal phase of the equipment, generating an impact on environment. Within this information we redirect energy issues to the direct natural resources impact. Hence, it is time to focus in carbon and emissions resulting in some studies [4] which have shown us up that IT is responsible of more than 2% of world total CO_2 emission, the first reason of the global warming. In addition IT hardware introduces other important problems both during its production and its disposal. We have to face the strong impact technology is creating on the environment and try to propose solutions able to reduce or eliminate its growing.

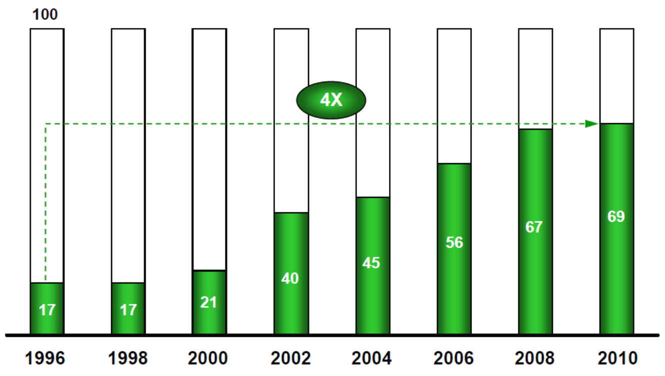


Figure 2.1: Ratio between New Equipments and Energy Consumption

Moreover, as verified in Figure 2.1 [5], the hardware costs has been increasing slowly for 14 years and in the other hand costs referred to power and cooling have been raising four times the equipments cost. Currently, all the costs of power required to be provided to IT infrastructures are almost 70% of overall IT costs. Forecasts say that the proportions will continually change in this way, since governments and power suppliers (worried with sustainable development and power consumption education) are planning to create special fees for periods when the consumption is generally higher during the day.

2.1.2 Initiatives and Solutions

The first occurrence of Green IT was in 1992, initiated by U.S. Agency responsible by a program created to promote energy efficiency in electronic equipments. From this program, some new resources started being adopted in systems design such as Sleeping and Stand By mechanisms to save energy.

Also, a certification program created by a Swedish organization called TCO started evaluating equipments related to its magnetic and electrical emissions like Cathode Ray Tube display and later every kind of energy consumption components. In 2006 the certification program created in 1992 established a strict layered ranking for all computers components and consequently the products which use such electronic parts.

The governmental side has begun focusing on Green IT during the last few years. Possibly the most important is the Climate Savers Computing Initiative. This non-profitable organization has groups of people responsible of publishing of power efficient new technologies. Furthermore there are some manufacturers participating of the mentioned organization. Their main objective is the reduction of computer power consumption to approximately 50% of the current consumption by 2010, which means a reduction of almost 54 millions of tons of CO2 each year. In the same field, there is the Organization called Green Computing Impact Organization which aims users guiding into a friendly behaving regarding environmental aspects related to electrical and electronic products. With the acronym GCIO, it organizes events in order to accomplish educational goals. This goal is considered the most important part of this organization.

Another respected organization is the Green Grid with more practical objectives, in other words, it has the target in optimization of overall IT products trying to increase their energy efficiency. The way it goes through the attempt to increase efficiency is the metrics and standards adoption to measure Data Centers energy levels for data analysis and further identification of feasible reduction points. Big multinational companies already joined Green Grid: HP, IBM, Sun, Dell and Microsoft. With the recent statistical information regarding this organization, it is the more promising green organization of current days.

From the participation of those organizations and consciousness of manufactures some solutions and innovative ideas have started to appear. Well show in following paragraphs some important ways to which Green IT issues have been addressed.

Computer virtualization is a creation of multiple computer resources into a single hardware. It is usually created using modified Operational Systems running into management software in attempt to use the maximum of the resources avoiding the existence of several physical computers underloaded. The guest software or operational system runs as if it was installed normally into a single physical platform in a transparent way to the user. Widely used nowadays, it requires a good knowledge in performance requirements and resources configurations. It must be known the availability of the resources that software needs to verify if the whole amount of software installed into a single physical platform will not compromise the quality of service provided. Virtualization, therefore, offers various advantages. Many physical servers can be replaced by a larger one, in order to increase the utilization of costly hardware. A virtual machine can be more easily controlled and inspected from outside than a physical one. New virtual machine can be created without the need to purchase more equipment. Virtual machines are just files and this allows the easy migration among different physical machines. Hence, virtualization is very important in Green IT since with it is possible to combine several physical systems into virtual machines hosted in just one single powerful system, thereby unplugging the original hardware and reducing powering and mainly the cooling consumption.

Terminal Servers in last years have also been used to put the computing operations only in the central computer, while terminals connected to it are simply thin clients and text terminals with low power consumption. This solution offers advantages in security and availability, since with a terminal breaking the service can still be provided. Advantages can be found out in terms of energy optimization. In fact, thin clients use no more than 10% of a regular workstation.

The Power Management, also sometimes known as Advanced Configuration and Power Interface (ACPI), is an open standard which defines ways to allow operation system to directly control the power saving aspects of underlying hardware. The OS can turn off autonomously a set of allowed components such as monitors and hard drivers during the inactivity period. The hibernation and stand by modes do use this standard. There are also programs to give users the possibility to configure the CPU voltages where the user will reduce powering, frequencies and also heat dissipation. Modern CPUs are able to configure it autonomously without users intervention.

Power Suppliers are usually 70% efficient with 30% of energy spent in heat dissipation. This means that for each 70W generated the supplier requires 100W in input. Hence this aspect became another topic to be addressed by Green IT. Higher quality power supplies can be over 80% efficient. There is less energy wasted in heat and surely less power is used in cooling. The industry initiated the 80 PLUS which is a initiative to building of powering units with more energy efficiency. It certifies products using some different parameters of load and power factor. From July 2007 all certified desktop computers are guaranteed to work with a maximum of energy waste of 20% of all powering capacity.

Other addressing targets of Green IT are Storage Units. Hard disk drives are the more mechanical equipments used in personal computers for example and requires a good amount of energy to work. Moreover, for instance, smaller drives (2.5 inches) consume less energy than larger ones (3.5 inches). Solid state drives consume even less energy, since their do not need power to mechanically squeeze the internal disk. These last types of drives use flash memory of a DRAM to store the data reducing power consumption. The trade-off in this case is the performance because is known that flash based drives are usually slower than the traditional hard disk drives.

Finally other important topic covered by the hall of Green IT Solutions are the displays with sensitive enhancements in last few years. Displays have slashed their power usage from an average value of 110W to 40W in last decade representing a reduction of more than 60%. This data was feasible since occurred a migration from CRT monitors to LCD and also because the prices of these new technologies dropped down very fast.

2.1.3 Focus in Software Layer

The Figure 2.2 presents the layers involved in IT structure and the proportional power consumption of each part used in full architecture. For a total of 100W spent with Data Centers system powering, approximately 60% is used in the server machine and from this value is extracted around 42% with processor summing its idle and calculating states.

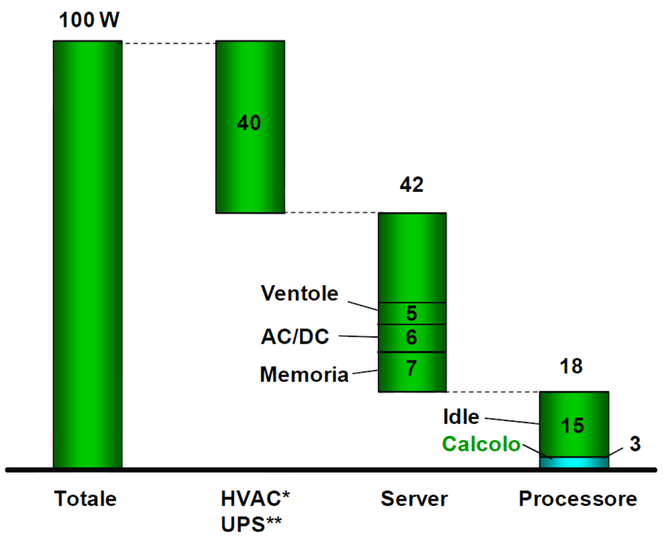


Figure 2.2: IT Layers and Proportional Consumptions

This work will focus in the software layer related to large Data Center, for example, from financial institutions. Must be known that even the Figure above showing that the software may handle a little amount of power consumed comparing to every part in architecture, the software has a dramatic interference in the final value because it defines the relation between idle and busy states of processor, and also can measure the performance requirements. These requirements are able to address the resources which must be available to the system, and also address possible virtualization.

From the physical point of view, the mean power consumption of a processor running any application is:

$$P = I * Vcc \tag{2.1}$$

Variable P is the power, I is the mean current used and Vcc the powering voltage. Then, the energy consumed by the processor running software is the integral of Power in function of time, verifying:

$$E = \int I * Vccdt \tag{2.2}$$

Therefore, to measure the energy consumed by the system must be known Current and Voltage powering the system. Moreover, its obvious that only the measure of these two components also does not guarantee the knowledge of each consumption points of the system. A deep analysis measuring each part separately can show up the specific consumption proportioned by running processor. From the logical perspective can be inferred from Margolous Levitin theorem [6] that the maximum frequency in state commutation from a physical system is directly proportional to the whole energy of the system. So, the minimum amount of commutation energy needed by a system to work properly in a certain frequency is calculated with:

$$E_{min}(f) = f * h/4 \tag{2.3}$$

In the equation above, f is the frequency and h the Planck constant. Now, going deeper, we reach other variables important in energy measuring. One of them is the systems information set measure used to define the amount of information need by it (i.e. number of bits). Also, we have entropy which is a level of disorder of data which can be transcript as the computational complexity for a desired output to be generated by a system. These two last components respectively Cc and Td, generate the Energy Consumed formula:

$$E_{logical}(f) = E(f) * Cc * Td \tag{2.4}$$

Its clear that there is a trade-off between frequency and energy where one increases the other runs after. Therefore, with the metrics and presentation of how information and complexity have influence in energy measurements is possible to clearly address the focus of this work, which consists of a set of software optimizations aiming the reduce of computational weight and consequently the energy required by the system to accomplish its tasks.

2.2 Software Profiling

Profiling the software is a resource available which allows the stakeholder to verify where a program spends its time and which functions are being called while the whole system is executed. This information can show which pieces of the program are slower than is expected and those that might be candidates for rewriting or substitution to make the program execute faster and save power and time. Moreover, other output is the presentation of how often certain functions and procedures are being called for execution. This may help to spot code blocks interesting for deep analysis.

Since the profiler uses information collected during the actual execution of the program, it can be used on programs that are very large and/or complex to analyze by reading the source directly. However, the way the program is running will affect the information that shows up in the profile data. Thus, is mandatory that all the important features and those with high performance influence are stimulated to provide reliability and usability of profile information generated in profiling analysis. Software profiling has some generic steps:

- Must be enabled profiling and/or the program must be compiled/linked with profiling support;
- The program must be executed in a desired way to generate relevant profile data;
- Must be executed a tool to interpret the generated data.

Depending on the language or platform (i.e. Java, C, C#) the steps above might be joined or transparent for the user. Profilers use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, instruction set simulation, operating system hooks, and performance counters. Summary profile information is often shown against the source code statements where the events occur, so the size of measurement data is linear to the code size of the program. In contrast, the size of a (full) trace is linear to the program instruction path length, making it somewhat impractical. For sequential programs, a profile is usually enough, but performance problems in parallel programs (waiting for messages or synchronization issues) often depend on the time relationship of events, thus requiring a full trace to get an understanding of what is happening. Our work is based mostly on sequential programs, focused in heavy mathematical/financial methods.

2.3 Financial Software

Nowadays, several financial institutions use complex systems to manage all its resources, including fees, taxes, accounts, investments and prices in general.

These systems are well-known as heavy computational operations including libraries and routines used to calculate various different values related to personal and corporate economies and accounting at all. They are complex because use math with exponential, rational numbers and series. Obviously, as the most of managerial systems, a set of variables and changeable parameters are available to the user, and this turn the servers into operations which if not run in high performance equipments may take several amounts of time.

This work will focus in software layer of Green IT field and furthermore, will take place in systems related to financial operations and calculations. Analyzing operations and using software profiles to verify and identify routines will be possible to optimize such executions, release idle resources and improve the available resources.

Initially the work will take place in analysis of a set of functions traditionally used in system to manage credit and insurance companies. They are:

- Return on investment (ROI): is the ratio of money gained or lost (whether realized or unrealized) on an investment relative to the amount of money invested. The amount of money gained or lost may be referred to as interest, profit/loss, gain/loss, or net income/loss. The money invested may be referred to as the asset, capital, principal, or the cost basis of the investment. ROI is usually expressed as a percentage rather;

- Future Value (FV): the value of an asset or cash at a specified date in the future that is equivalent in value to a specified sum today;
- Present Value (PV): is the value on a given date of a future payment or series of future payments, discounted to reflect the time value of money and other factors such as investment risk;
- Net Present Value (NPV): a time series of cash flows, both incoming and outgoing, is defined as the sum of the present values (PVs) of the individual cash flows.
- Duration : the duration of a financial asset, specifically a bond, is a measure of the sensitivity of the asset's price to interest rate movements. Duration is known in the context of the "Greeks" used for derivative pricing as the Lambda.In contrast, the absolute change in a bond's price with respect to interest rate (Delta) is referred to as the dollar duration. The units of duration are years, and duration is generally between 0 years and the time to maturity of the bond. It is equal to the time to maturity if and only if the bond is a zero-coupon bond. Duration indicates also how much the value V of the bond changes in relation to a small change of the rate of the bond.
- Convexity : Convexity is a measure of the curvature of how the price of a bond changes as the interest rate changes.Specifically, duration can be formulated as the first derivative of the price function of the bond with respect to the interest rate in question, and the convexity as the second derivative.

- XIRR:calculates the annualized internal rate of return of a cash flow at arbitrary points in time.Values lists the payments (negative values) and receipts (positive values) with one value for each entry in dates.

- Binomial Option Pricing (BOPM): The model provides a generic numerical method for the valuation of options. Essentially, the model uses a discrete-time model of the varying price over time of the underlying financial instrument.

- Implied Volatility :The implied volatility of an option contract is the volatility implied by the market price of the option based on an option pricing model. In other words, it is the volatility that, when used in a particular pricing model, yields a theoretical value for the option equal to the current market price of that option. So, this function returns the implied volatility by a given option and its parameters.

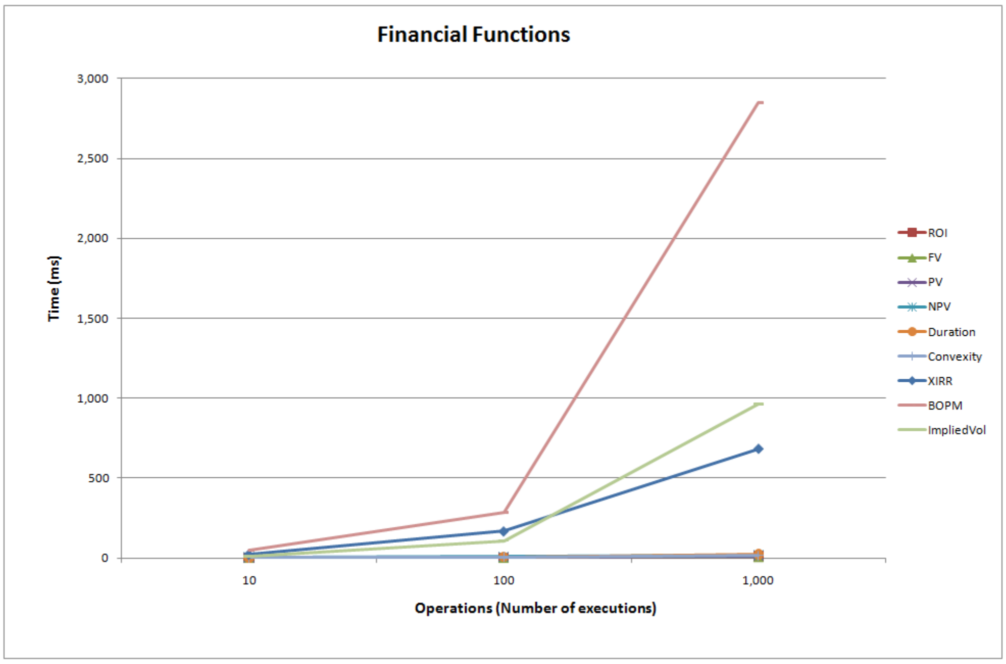


Figure 2.3: Time Execution for a low number of operations

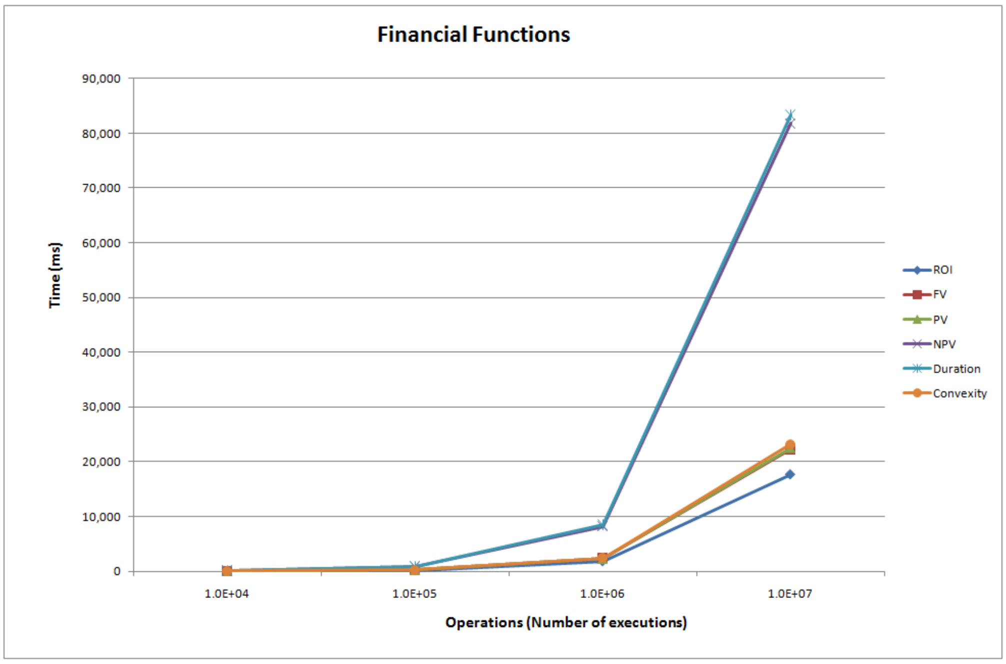


Figure 2.4: Time Execution for a high number of operations

Those calculations presented on previous page are also considered the algorithmically bottleneck of financial enterprise systems, compromising overall performance. In fact, to attack those points, it will be necessary to identify the pure functions, in other words, mathematically pure functions (functions without any other functionality but math, no output, no interface, no event handling and no classes attributes changing).

The study regards some substitutions of series calculation by Tables with pre calculated values depending on chosen parameters to optimize time of financial software execution.

An example for execution time behavior of some financial computations is presented in both Figures 2.3 and 2.4. The difference between the charts is the number of operations range.

Chapter 3

A Green Software Approach

The project developed in this thesis consists of a deployed Java software modification to improve performance in mathematical operations by caching often used values in memory instead of reprocessing them every time. More specifically, this thesis concerns about a module of data storage (for caching) and also a module of decision (when caching). The target companies to be used as use case for the development will be financial companies such as banks and insurance providers. Their software commonly have a large set of mathematical functions to calculate taxes, fees and prices (i.e. Stock Market).

Within this mathematical environment, was defined a Green approach of power saving to these companies with the idea to modify the running systems to fit the saving of energy consumption. The target is the set of pure functions, already explained previously. Since there are various different students involved in analysis, development and testing the project is modularized.

The first step of the project, running on the same time as other modules development, is the definition of which methods are capable of modification without any damage to the whole system behavior. Since the software language is Java, there is bytecode tools useful for code behavior identification. The methods must not have any other operation than the mathematical calculation and nor change any class attribute during its execution, as described in pure function definition.

The following step is the generation of a new set of operations to substitute the original one. This new set of operations is the, for instance, the call for the lookup table to verify if that set of parameters are already stored in the cache system. If not, the system will have the mathematical original code to be executed to generate a result, and then, will call a module responsible to keep the data in cache if the constraints are satisfied. All the architecture modules and constraints will be clearly explained in sections below.

Bearing this in mind, is possible to affirm that the system working for a long time has a trend to generate a great saving in processing time and consequently in power saving, since the processor will not be charged with heavy operations every time. After successful testing, the adapted system will be deployed to the production environment.

3.1 Software Architecture

The architecture of tabulation and decision modules is described in Figure 3.1. The first module, called Decision Maker, is an abstract module which represents the Byte Code modification applied to the original system. The Decision Maker itself is the function and the sequence of Lookup / Processing / Trade-Off execution flows. The following two sections explain in details the functionality and behavior of Memory Management and Trade-Off modules, respectively.

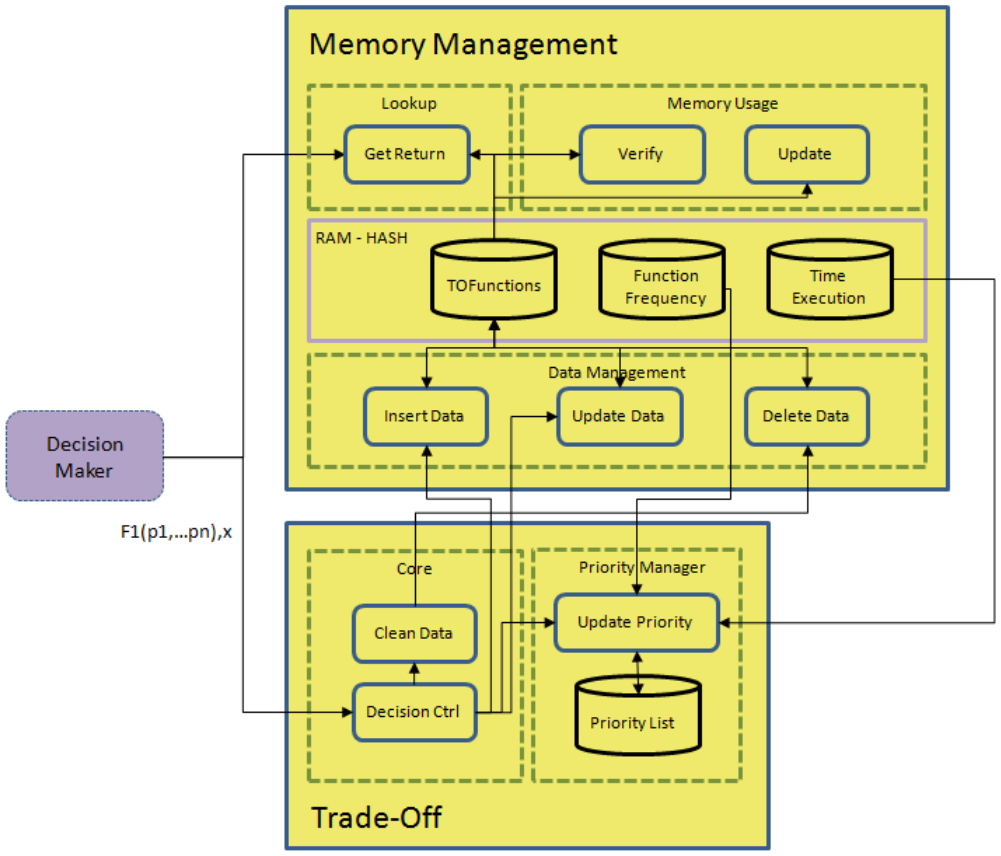


Figure 3.1: System Architecture

3.1.1 Memory Management

This is the module responsible to operate the data inside the pure functions cache systems. The module is composed by a set of memory control, such as Total Memory, Free Memory, Lookup Status and the Storage Structure using a `HashMap`. The `HashMap` has a key of String type with information related to Package, Class and Method names with also the Java Signature. This is a unique key to identify the functions. The `HashMap` has values in `TOFunction` format, described in details in Figure 3.4. All useful data will be stored inside the `TOFunction`.

As a typical Memory Management module, it uses the Singleton Pattern which makes available to the system always the same object instance to guarantee consistency. Also provide an API to the whole system to execute data operations like insertions, deletions and updates. For the abstract module Decision Maker, the main operation and the most used feature is the lookup, where the pure functions will look for previously tabulated values prior to mathematical computation. Moreover, are provided methods for sizing computation, which allows other modules to verify whether a data will have enough space to be kept before trying to execute the insertion itself.

3.1.2 Trade-Off

The main role of this module is to manage the data inside the Memory Management accordingly to defined logic and rules. The module Trade-Off is responsible to provide the intelligence of decision regarding the recording or not of some specific pure function information (parameters and result). It provides the API to be accessed by the Decision Maker abstract module and to access the Memory Management singleton. This module keep a priority reference of each `TOFunction` stored in memory and is deeply used when removing of previous tabulated functions is required by the absence of free space in memory.

The submodule of Trade-Off called Priorities Manager makes (when executed the method `UpdatePriorities`) the automatic updating of the priorities reference by consulting the available API provided by the Memory Management module. Actually, the priorities are not stored in Memory Management because this logic is not relevant to Memory Management. Since Trade-Off is the intelligence, it is responsible to handle all priorities. Priorities are explained in a finer way in the subsequent sections.

Function Level Priorities

The function priority is a way to split the free memory among the function data spaces required, trying to decide in an optimized way the amount of points allowed to be recorded to each function and how to prioritize one function instead of another. The priority is obtained using the formula:

$$Priority(f) = \frac{N_{calls}(f) * T_{exec}(f)}{\sum(N_{calls}(fi) * T_{exec}(fi))} \tag{3.1}$$

Value Level Priorities

The Value Level allows two different approaches to determine the `TOFunction` individual priority. The first and simplest one is using the Frequency of each entry considering Lookups for that value and also its computation. Another provided method, and a computationally heavier one, is when the Value priority is known as the point score and it gives the opportunity to compare different points based on the existing data, working as a clusterization algorithm. Whenever it achieves the equilibrium it is expected to have, as chosen values to compound the value table of a function, the set of points near the inputs average. The mathematical formula to compute these values is the Mahalanobis distance:

$$d(x,y) = \sqrt{\sum_{i}^{N} \frac{(xi - yi)^2}{\sigma^2}} \tag{3.2}$$

The practical usage of the method will depend on the performance verified during preliminary execution tests.

3.2 Memory Organization

In this section is described how the data (most important functions inputs/output) will be kept in the memory structure for cached values during execution of the system. In Figure 3.2 is shown the organization of the `HashMaps` used to fast access of whole data.

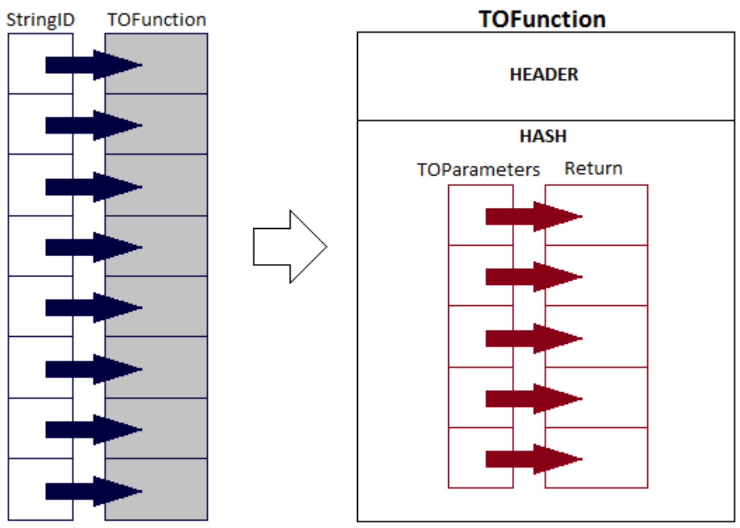


Figure 3.2: Memory Schema to keep the function data

3.3 Use Cases

In Figure 3.3 is presented the main usage methods of the module developed for this thesis. The most important action - operation available to the actor (Decision Maker: a conceptual module present in the whole project architecture) - is the Trade-Off. The action consists of an intelligent algorithm which determines whether a pure function of the financial system and its entries (parameters and return) will be kept in memory, characterized as a cache memory, to improve the

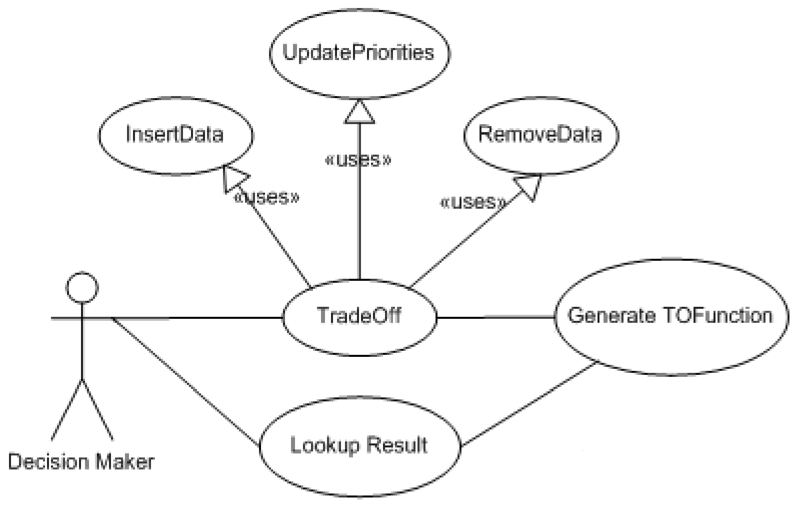


Figure 3.3: Use Case Diagram

relation between computing and power consumption. Since the computation will be reduced by this storage mechanism, the consumption of the processor and related electronic components will reduce.

The Trade-Off uses some operations provided by the Memory Management like insertions, updates and deletions: `InsertData`, `UpdatePriorities` and `RemoveData`, respectively. After verifying the logic and deciding the including or not of a function into the memory, the Trade-Off is responsible to operate the memory sending requests to make the memory module handles the data. These requests will keep also the memory state (Full, Available) and will provide methods to remove functions with lower priorities to save new data.

Lookup Result is the second most important use case. This is a simple and easy set of steps to consult a required data in the memory. The function receives a function structure with parameters - from the Decision Maker - and is responsible to follow the `HashMaps` and return a miss if the data is not present in the cache and the system will need to execute the whole mathematical method to compute the result.

The creation of the `FunctionSignature` is made by the Decision Maker and this signature will be later translated to a simplified structure to keep in memory aiming the optimization of the spaced used by the function data storage. This whole scenario describes the architecture involved in decision of data keeping and will be detailed in following sections presenting sequence diagrams.

3.4 Classes Diagrams

This section will present all classes used in the software architecture created for this thesis. Each package subsection will present a graphical presentation of the classes and then a brief explanation about the methods and attributes.

3.4.1 Utils Package

The utils package 3.4 has the common business rules classes to support both Trade-Off Decision and Memory Management modules. Must be known that all classes used in trading operation have TO prefix in its name. This is the acronym of Trade-Off. Thus, the most important attributes and methods of the classes are described below:

1. **Return**: used to store the original function result;
   - pValue: reference to the return object (Integer, Double,. . .);
2. **TOParameters**: used to store and manipulate the set of input parametes;
   - mHashCode: keep the calculated `HashCode` since the values will not be changed during execution and this improves `HashMap` operation speed;
   - input[]: vector or objects to make reference to the set of input parameters of the function. This structure allows the use of at most one more array level. In other words, this means that each position of the input vector supports the reference of more a vector. This limitation is important because the content of the vector and its aditional dimension is used to compute the `HashCode` for further `HashMap` processing;

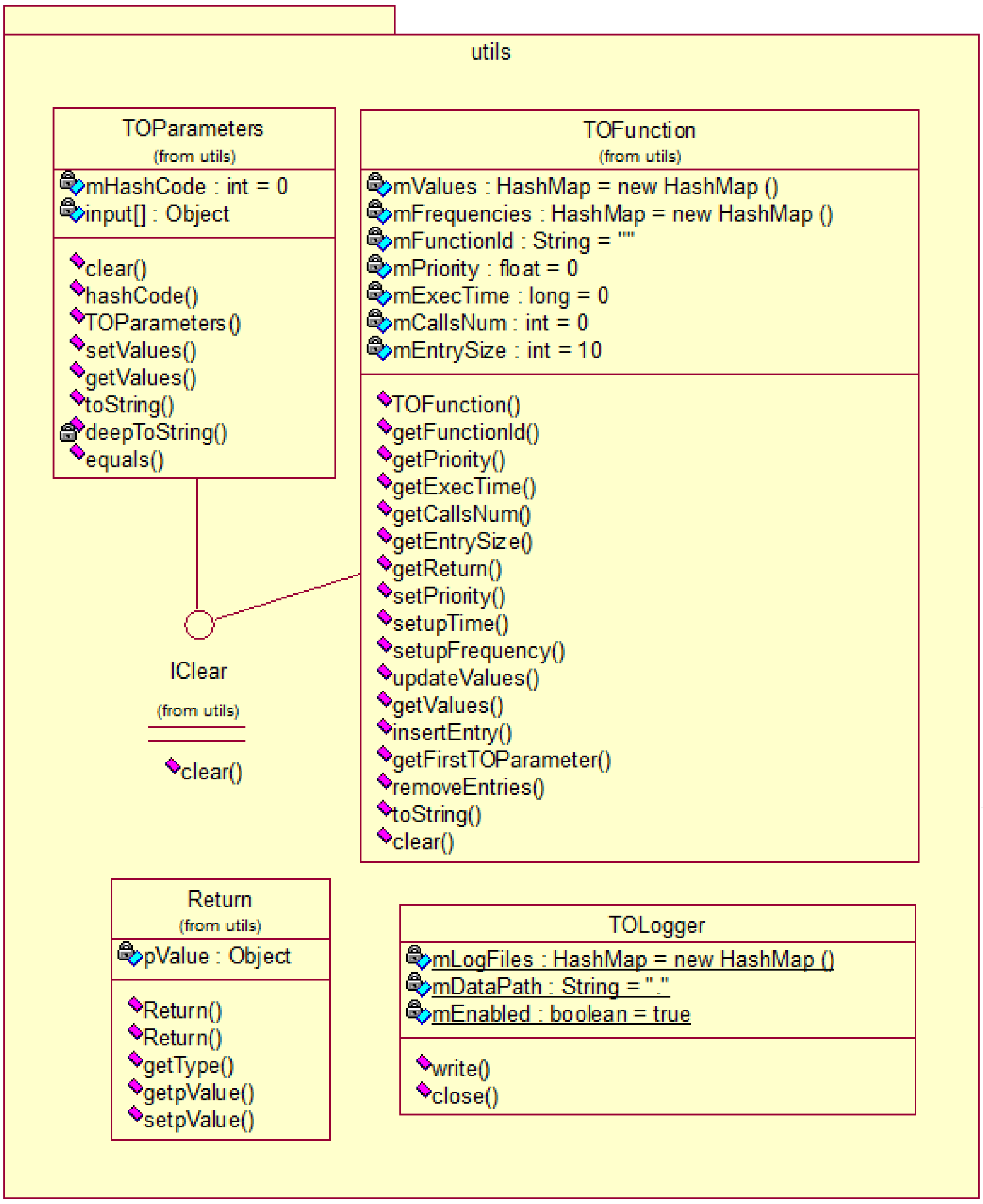


Figure 3.4: Utils Package - Important Classes

- *hashCode()*: overridden method to compute the `HashCode` by the convertion of all input[] content into String to guarantee the uniqueness;

- *toString()*: creates a String with the input[] content;
- *deepToString()*: processes the aditional dimension of the input[] content;

3. **TOFunction**: used to keep the function information and its values;

- mValues: a `HashMap` containing all the pairs `TOParameters` and `Return` related to that function. The `TOParameters` is the key object and `Return` the value. `TOParameters` has the `HashCode` overridden, since it has to behave like a key;
- mFrequencies: a HashMap to keep information about the frequency of each `TOParameters` and `Return` pair. The key of this map is the same of the mValues (`TOParameters`) and the value is an Integer. This frequency increases following the number of calls made to that `TOParameters` by means of the Lookup operation executed in the Memory Management module;
- mFunctionId: the attribute used as key in the Memory Management `HashMap`. This is a unique identification to each function tabulated. Format: ;package;class;method;java_signature;
- mPriority: this value represents the priority of that method in relation with the all others being processed to be tabulated or not. The calculation of this field is explained in the Priority Section of this document;
- mExecTime: field to save the execution time of the method. The execution time consists of the running of the original mathematical code of the function. The value is in nanoseconds;

- mCallsNum: this is the sum of all frequencies of all entries in the mValues `HashMap`. In other term, this is the total number of calls the function had, be with HIT in lookup, be with mathematical calculation;
- mEntrySize: this attribute has the size of each `TOParameters` and Return stored inside the TOFunction. This attribute is very important to control the amount of memory being consumed to tabulate each function. The value must vary according to the number of input parameters of the function and the type of the `Return` object;
- *TOFunction()*: constructor;
- *getFunctionId()*: returns the unique identification of the function;
- *getPriority()*: returns the priority;
- *getExecTime()*: returns the execution time in nanoseconds;
- *getFirstTOParameter()*: since the TOFunction is used every execution that must call Trade-Off module, when this situation happens, the TOFunction will contain exactly one entry in mValues attribute. And this method is used when the value must be inserted in the cached TOFunction structure, by copy operation;
- *setupTime()*: update Execution Time;
- *setupFrequency()*: update Frequency of an individual entry. Also updates the total frequency (mCallsNum) of the function;
- *getValues()*: return the reference to `HashMap` with values;
- *removeEntries()*: used by the Memory Management module to clean

data to get space to a higher priority function be tabulated. This function receives the number of entries to be cleaned from a specific function table;

- *insertEntry()*: insert a new `TOParameters` and `Return` pair to the function's table;

4. **TOLogger**: a simple class to log the execution of the system;

   - mEnabled: a configurable attribute to activate or deactivate the logging operation (file writing);

3.4.2 Memory Package

The Memory Package 3.5 consists of a unique big class. It uses the Singleton pattern to provide a unique instance to any call to it since it must guarantee consistence of the data stored. Below the most important information of Memory-Management class is mentioned:

1. **MemoryManagement**: used to store the original function result;

   - mMemory: a `HashMap` containing all the structured data kept from Functions and its values (`TOParameters` and `Return`). This `HashMap` has a key of String Type. This key is the field mFunctionId of the TOFunction class. The value is the full TOFunction object with all relevant information;
   - mTotalMemory: configurable attribute to set the maximum size used to store the data from the functions;

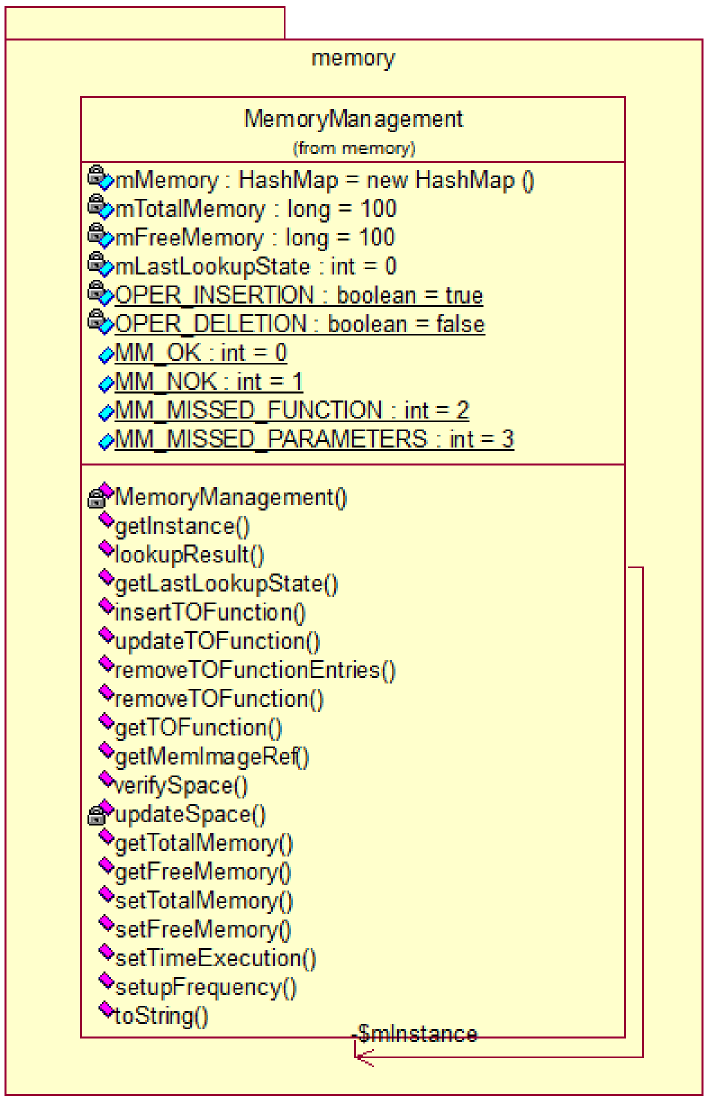


Figure 3.5: Memory Package - Memory Management Class

- mFreeMemory: a internal control to determine how the memory is being used by the data storage;

- mLastLookupState: a cached status to facilitate to discover which operation was executed before. If there was a miss in a function (the function is still not inserted in tables) or an specific pair `TOParameters` and `Return`. This is used for further Trade-Off execution;

- *MemoryManagement()*: private constructor;
- *getInstance()*: the method used by system to retrieve the unique instance. Method specified by the Singleton Pattern;
- *lookupResult()*: try to retrieve a already stored `TOParameters`. If there is any, return a `Return` object, otherwise, Null;
- *insertTOFunction()*: inserts a function to tabulation. It is possible to exist functions stored with no values inside, because the system must keep the frequencies and other function relevant information in order to generate statistics of all tabulable functions, even with low priorities;
- *updateTOFunction()*: makes the target TOFunction update its values with the new one passed in parameter;
- *removeTOFunctionEntries()*: removes an specific number of entries of a TOFunction to liberate memory space;
- *getTOFunction()*: returns the TOFunction related to that String ID;
- *getMemImageRef()*: returns a reference to the `HashMap` containing all information of memory (mMemory);
- *verifySpace()*: method used to verify if the TOFunction passed as parameter would have enough space in momory to be kept in internal table;
- *updateSpace()*: private method to update the memory space control after a determined operation with TOFunction Entries - removal or

insertion;

3.4.3 Logic Package

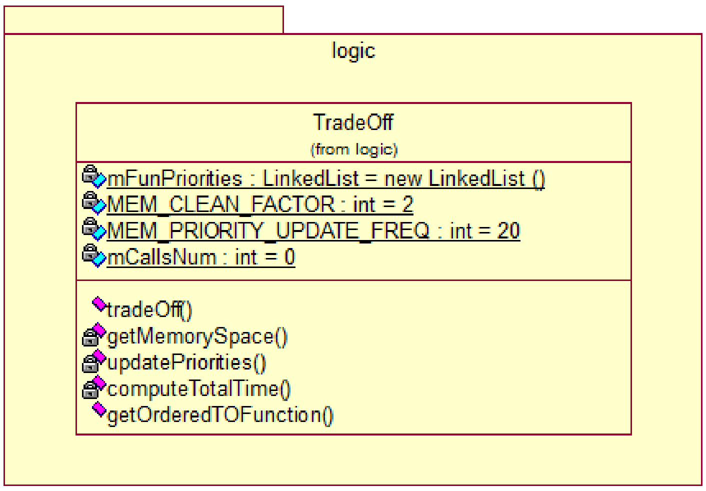


Figure 3.6: Logic Package - Trade-Off Class

The Logic Package 3.6 is composed by a class called TradeOff, responsible to process all the logic regarding the data saving, priorities manipulation and memory verification. Following are cleared some aspects of the presented diagram:

1. **TradeOff**: class with the decision logic of storing or not cacheable information;

   - mFunPriorities: this attribute is LinkedList ordered by TOFunction priority. The content inside each position of the list is the String ID used to reference a certain TOFunction inside the Memory `HashMap`;
   - MEM_CLEAN_FACTOR: important configuration of the Trade-Off module. This values determines the amount of memory to be cleaned before inserting a new information (with higher priority) in case of

absence of space for a new insertion. This value is actually a multiplication factor. If the new entry requires 10 bytes to be cached and the attribute value is 4, then the TradeOff module will clean 40 bytes. This value is important to avoid a removing operation in every attempt to store new information in memory;

- MEM_PRIORITY_UPDATE_FREQ: another important performance configuration. Must be known that the operation of Priority List (mFunPriorities) is a heavy operation and is unfeasible to be executed every time. Then, this attribute control to the system update the priorities list only every N accesses, where de N is the number set of this class field;
- mCallsNum: auxiliary variable to control the priorities updating explained in the item above;
- *tradeOff()*: the main function of the TradeOff module. A TOFunction is passed as parameter and the module decides whether to store it or not;
- *getMemorySpace()*: method used to logically liberate space in memory for a higher priority function be kept. The method starts by the lowest priority, iterating and removing until the defined amount of memory (see MEM_CLEAN_FACTOR) be free;
- *updatePriorities()*: private method to update priorities;
- *computeTotalTime()*: private method to update priorities;
- *getOrderedTOFunction()*: this method returns an Iterator to navigate through TOFunction String Ids following priorities ordering;

3.5 Execution Sequences

These sequence diagrams were built in order to provide an idea of internal module interaction. The Decision Maker module is the process trigger, starting with a table lookup. That will use the method provided by the Trade-Off module to obtain any point (result) needed by the system. The input is the object `TOFunction`, which contains all data used by the module to recognize the function and the parameters.

After asking a specific point, the Trade-Off starts fitting the pure function data into its own objects. The first step is to generate a function key that will be used to distinguish different functions and types required. After having the key it is possible to identify the function but not a specific point inside the values table. So, the role of the `TOParameter` is to hold the parameter objects with the purpose to tell apart among all the points on the value table. At this point of the sequence all the information needed is already translated and the Memory Management will check if there exists the value for this parameters. If yes (Hit) the value is returned and the sequence is over and if not (Miss) it is returned the value null.

The second step of the sequence is started if the Decision Maker receives a miss from the lookup method. Whenever it happens, this point is calculated and passed to the Trade-Off module. In this case, an object `TOFunction` will be instantiated to keep all data and the free memory will be checked. If there is enough room, this point is added to the value table and the value 0 is returned to the Decision Maker.

Supposing all the memory is full and the function is already tabulated. The score of that point is calculated and compared to the last score of the tabulated values, if the new point score is higher than it is discarded and the value 1 is returned to the Decision Maker, otherwise the worst value, in terms of score, is deleted and the new one is added returning the value 0.

Another possible scenario is having the memory full, but the function is not yet tabulated. In this case the priority of the functions is taken into consideration and compared to the new function, if the new function has a higher priority than the worst priority of the tabulated functions, the worst function is deleted and the new function is added. With this description all possible scenarios were explored and the sequence diagrams can be found below.

3.5.1 Lookup Scenarios

Scenario Miss

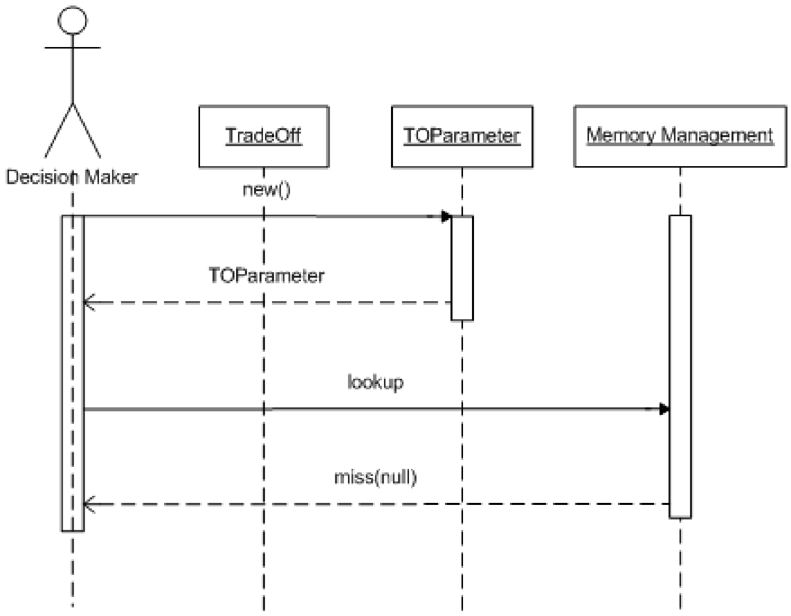


Figure 3.7: Lookup with Miss in Memory. Requires Trade-Off execution

In this scenario, presented in Figure 3.7, the modified pure function creates the object `TOParameters` with a vector of inputs (parameters) of the original function and then call the lookup method of the Memory Management passing two parameters: `FunctionID` (String) and `TOParameters`. The Memory Management singleton tries to execute a get in its `HashMap` of storage with the `FunctionID`. If the return of the get execution is `NULL`, then a function miss occurred and the lookup will immediately return. If the content was found in the `HashMap` with the used key (FunctionID), then the system looks for the entry composed by the parameters passed to the pure function. The `TOParameters` is the key of the `TOFunction` values `HashMap`. The same get method is executed. If `NULL` is returned, a miss in Parameters level occurred and then the lookup returns a miss.

The situation above required the execution to be continued inside the modified function. The mathematical computation will be done and then the Trade-Off function will be called to verify if the function just computed needs to be kept in memory or not. Thus, the continuation of this scenario and the various situations are found on the following Trade-Off scenarios section.

Scenario Hit

On the opposite way as shown in Figure 3.8, if any valid `Return` value was found given the successive keys (`FunctionID` for the first-level `HashMap` and then `TOParameters` for the `TOFunction HashMap`), the lookup executed a hit procedure and the Return value is sent to the modified pure function and no more mathematical computation is required to that case since the value was already present in cache structure.

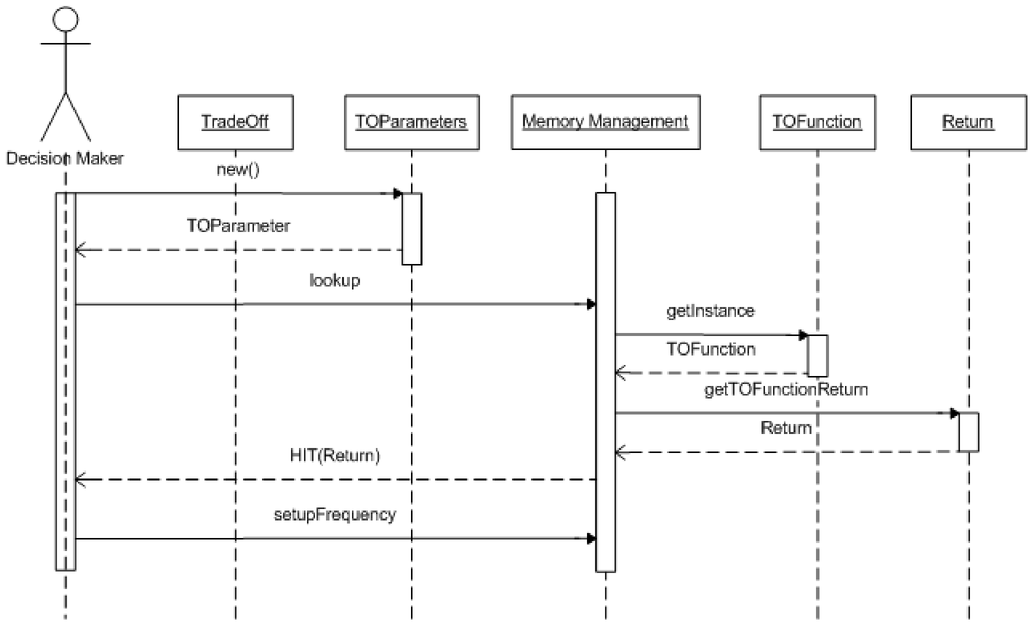


Figure 3.8: Lookup with Hit in Memory. Get the return and continues

3.5.2 Trade-Off Scenarios

The Trade-Off scenarios are mapped into a flow where in first steps the Lookup execution returned a miss, Figure 3.7: Function not tabulated or even a particular entry. In other words, the modified method tries to find the cached value for that specific execution. Then, if no value is found, it executes the normal mathematical operation and initializes the structure (`TOFunction`) to call Trade-Off. In the procedural sequence, the system will verify regarding priorities, frequencies and execution times whether the recording of tabulated value will save processor activities and therefore power. The following two sections describe each of possible situations.

Available Memory

The first scenario illustrated in Figure 3.9, and the simplest one, regards the execution of Trade-Off memory with available space in memory. Hence, it is not required to verify priorities before deciding about value recording or not. Basically, the modified function runs the mathematical procedure and computes the result. This result is stored in the structure Return and then used to create the TOFunction itself. The time spent to math execution is calculated for posterior priority purpose. After, the method calls Trade-Off.

The first execution step is the verification of the last lookup execution result. There are two miss types: function and parameters. The Function Miss indicates that the function itself is not in memory, so information like signature, execution time is not available. In this case the Trade-Off creates the function in memory (this function object with no entries - parameters and return - are not considered in memory consumption control since it can be neglected by the exiguous usage of memory comparing to the gross amount of entries stored inside the object, in `HashMap`). Later, the Trade-Off requires to Memory Management a calculation of memory availability based in the space required by the new function - remembering that each function has particular number of inputs and then may consume more ou less than another function.

In this case, the manager indicates available space and no cleaning procedure is required. The following step, so, is the function update. Relying on the `TOParameters`, the Trade-Off module calls the method `updateTOFunction` sending the `TOParameters` as parameter. The procedure inserts the new entry inside the `TOFunction`. Right after, the Trade-Off returns the operation result to the Decision Maker and the last step is the updating of frequencies by the calling of `setupFrequency`, where the system updates the number of calls of a particular `TOFunction` entry tabulated.

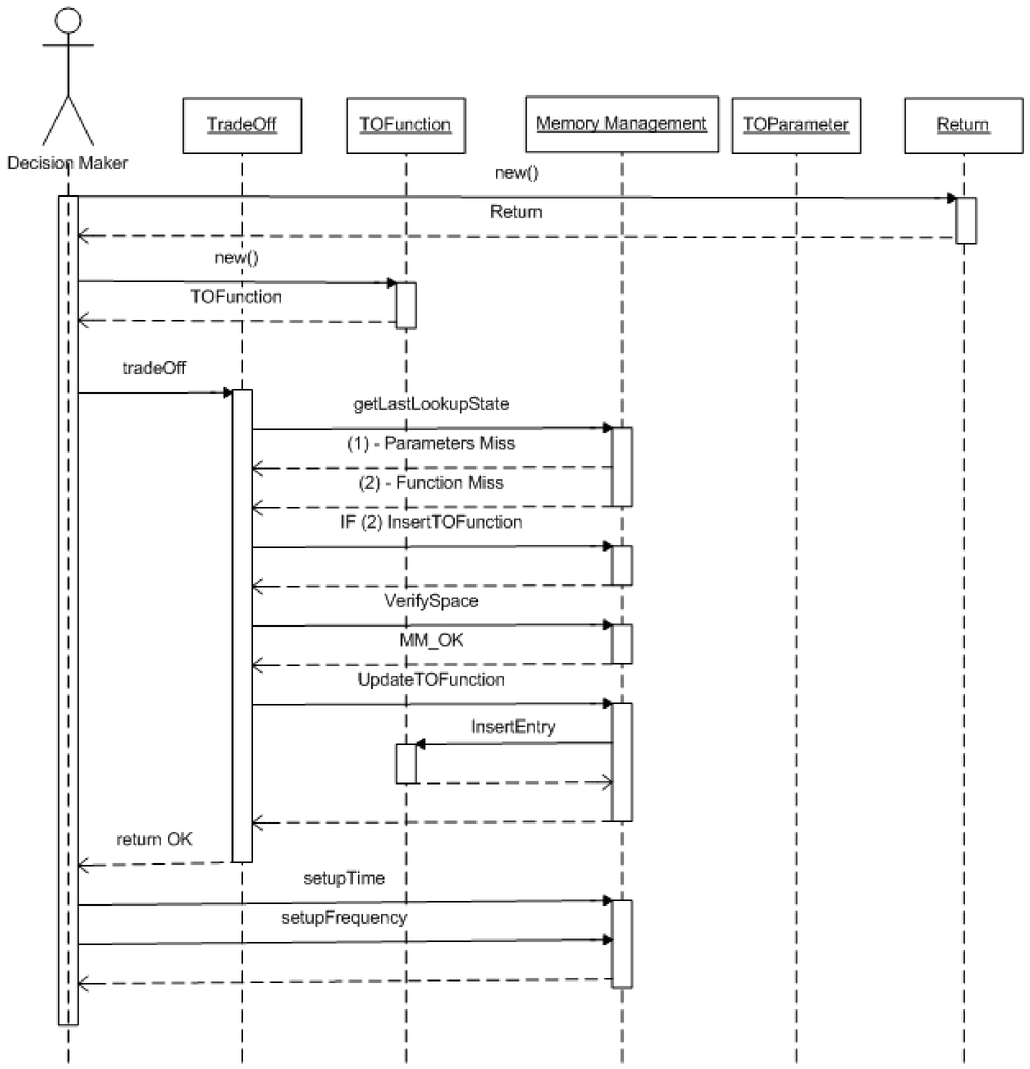


Figure 3.9: Trade-Off with memory available

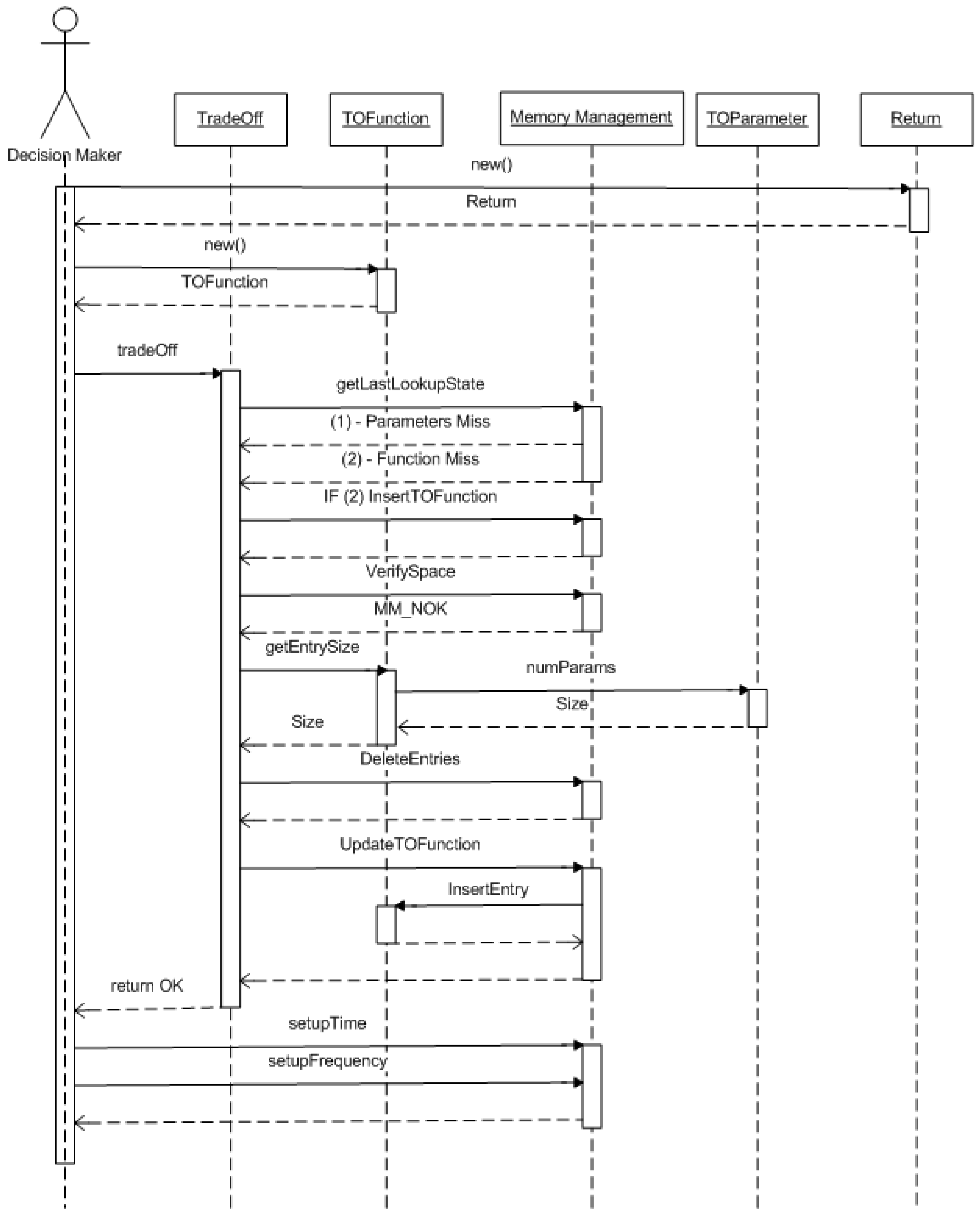


Figure 3.10: Trade-Off with full memory. Handle cleaning/Insertion

Full Memory

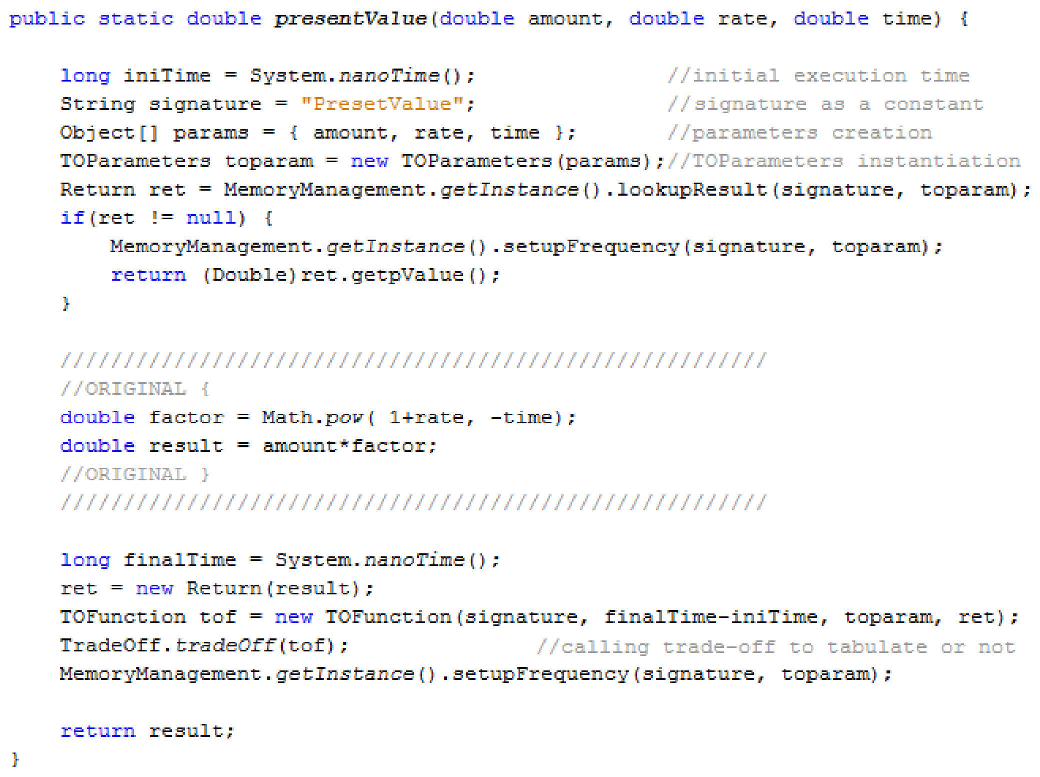
Comparing to the situation where there is available space, Figure 3.9, to keep data in memory, this scenario, Figure3.10, can be a copy of the mentioned situation with an intermediate execution step responsible to clean the memory for the new `TOFunction`, of course, if needed. The Trade-Off module keeps an ordered list with the `FunctionID` to control the priorities. When the function to clean memory is called, the ordered list is roam and the priorities with already saved function are compared to the new one.

If the priority of the new function is higher, the lower priorities `TOFunction` have entries deleted till the needed space in created. One important configuration resides in the memory cleaning operation: `MEM_CLEAN_FACTOR`. This value is multiplied by the required new `TOFunction` entry space and then is an operation that allows the system to create more space then really need to save performance. Is known that, is the memory is already full and the system have a granularity of a single entry of a single `TOFunction`, the performance would decrease drastically removing elements with priority control to each Trade-Off decision execution. Moreover, if the needed space is not fit with the deletion of all entries of a `TOFunction` with lower priority, the system will go to the next `TOFunction` with lower priority. The iteration will remain until the needed space is reached.

In a condition where the priority of the new function to be kept is lower than all other priorities already saved in memory, then the `TOFunction` is simply discarded and in case of a new execution of the same method with the same parameters a miss will occur in lookup execution and the pure function will need to recalculate the result. After the cleaning process, the execution calls the `update TOFunction` in the same way as the memory had available space, how was described in the Available Memory scenario.

3.5.3 Sample Source Code

To illustrate the sequences described above, there is a practical example of Java source code of how a modified pure function behaves. The Figure 3.11 presents the blocks responsible to each flow and each scenario, varying from a lookup with HIT result (the fastest execution possible) to the Trade-Off flow with no available memory to keep data (the slowest execution).



```java
public static double presentValue(double amount, double rate, double time) {

    long iniTime = System.nanoTime();                //initial execution time
    String signature = "PresetValue";                //signature as a constant
    Object[] params = { amount, rate, time };        //parameters creation
    TOParameters toparam = new TOParameters(params);//TOParameters instantiation
    Return ret = MemoryManagement.getInstance().lookupResult(signature, toparam);
    if(ret != null) {
        MemoryManagement.getInstance().setupFrequency(signature, toparam);
        return (Double)ret.getpValue();
    }

    ////////////////////////////////////////////////////////
    //ORIGINAL {
    double factor = Math.pow( 1+rate, -time);
    double result = amount*factor;
    //ORIGINAL }
    ////////////////////////////////////////////////////////

    long finalTime = System.nanoTime();
    ret = new Return(result);
    TOFunction tof = new TOFunction(signature, finalTime-iniTime, toparam, ret);
    TradeOff.tradeOff(tof);                     //calling trade-off to tabulate or not
    MemoryManagement.getInstance().setupFrequency(signature, toparam);

    return result;
}
```

Figure 3.11: Source Code of a Modified Function

Chapter 4

Power Optimization By Caching

This chapter will present to the reader all the logic definitions used in the development of the Memory Management and Trade-Off modules. This will explain in details how the tabulable functions are identified, how the memory usage is estimated and further features used for decision support.

4.1 Trade-Off Modes

The implementation of Trade-Off module provides two distinct operation modes: Full Tabulation and Range Tabulation. These both modes will be described more in details in sections below but the basic idea is to allow the users to have different way to decide whether to tabulate one information of not. This may improve the quality of stored data and also optimize the memory organization inside tables.

4.1.1 Full Tabulation

In Full Tabulation operation mode no tests will be performed over the entry data. In this way, the Trade-Off module will take into account only decisions support such as priority and time execution among all possible functions. In other words, all the functions modified by the abstract Decision Maker modules will have the same change to be stored depending on its priorities, independently of how spread or not are their data. Within this type of execution, the system may experience a kind of memory wasting seen that functions with high priorities might have a set of data stored in memory but those data will be seldom accessed by the Look Up execution.

This operation mode will be controlled by the attribute `private static int mLogicMode;`, inside the class TradeOff.java. The operation mode of full tabulation without discarding data according to mathematical tests is: MODE_KEEP_ALL. By this time, no interface is provided by the class to change this attribute, and it is configured inside the code. Dynamism is considered a next step to the system development, when enhancements will be incorporated.

4.1.2 Range Tabulation

In other hand, the Trade-Off operation mode can be set up as Rage Keeping by the TradeOff.java class attribute: `MODE_KEEP_RANGE`. In this mode, every Trade-Off execution will initially analyze the entry values and verify if they match the defined cut configuration. This execution flow will use three different configuration attributes concerning the modified method to be tabulated or not. They are applied to `TOParameters` values, in the way where the Trade-Off will verify if for determined function the parameters are according to the Range desired to be kept and the continue the module table insertion flow.

- Average: the average value to be tested with each `TOParameters` value and verify whether it's matching the definition. This value is independent to each value of `TOParameters` because the entry values are not correlated among themselves;
- Standard Deviation: these values count how spread values are going to be processed of not. This control will avoid values far from average to be kept in memory since they possibly will have low probability to be looked up again. This value is also parameter independent;
- Cut Standard Deviation Number: this value indicates how many Standard Deviations need to be considered, up and down, in values verification. This value is a generic configuration of internal Trade-Off module control. Once modified, this will affect all the parameters of all functions which use Trade-Off tabulation.

In Figure 4.1 is shown how works the configuration described in the enumeration above. In the image there is the function table that means the memory where functions and values are kept. Inside each of the functions (i.e. Present Value and XIRR) there are the values saved. The first rounded rectangle with three values represents the input parameters and the other with a single value represents the function output to the written inputs. The gray rectangle shows us up the XIRR configuration that is a defined array inside the modified XIRR function which defines the average and standard deviations considered to each value of input (TOParameters).

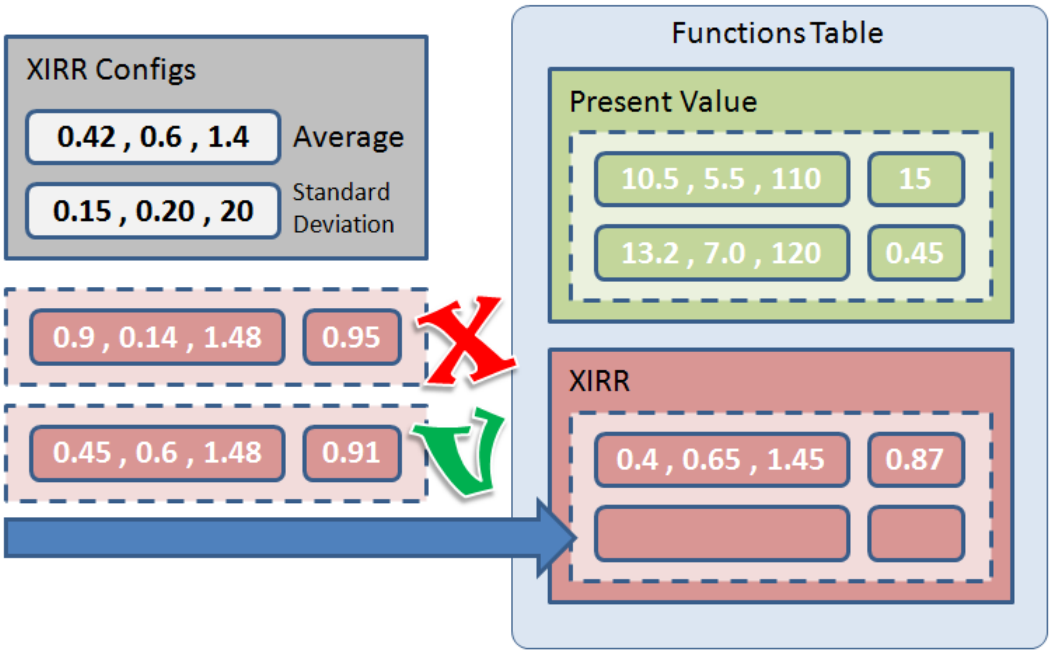


Figure 4.1: Trade-Off Range Mode Configuration and Verification

As mentioned before, beside the Average and Standard Deviation there is a configuration to define how many deviations up and down must be considered to process the range inside the storage table. In the example illustated in the Figure 4.1, we consider this value 2. By this way, we can verify that Trade-Off module will discard the first set of input values, because the first input 0.9 in out of the range 0.42 $\pm$ (2)x0.15 configured to XIRR function. The acceptable interval would be, in this instance, $0.12 \leq x \leq 0.72$. The rule applies to all input parameters independently.

Therefore, this verification will make Trade-Off discard the entries with no correspondence to the configuration as seen in the first input set of the Figure 4.1.

4.2 Alpha Approach

The Alpha Approach is the document session which will analyze the different code blocks and their executions to verify time consumption and create a model to evaluate how good for tabulating a function is and the breakeven point to save its data in memory or not. To illustrate and code blocks is presented a flow diagram in Figure 4.2.

4.2.1 Execution Flow

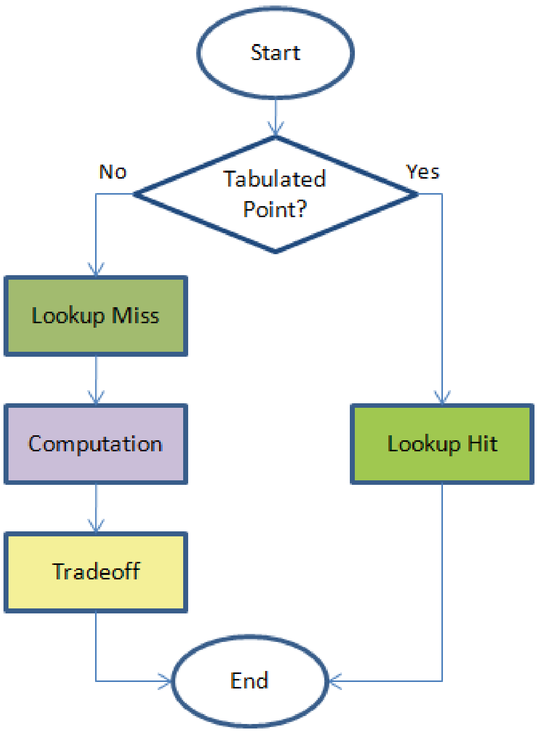
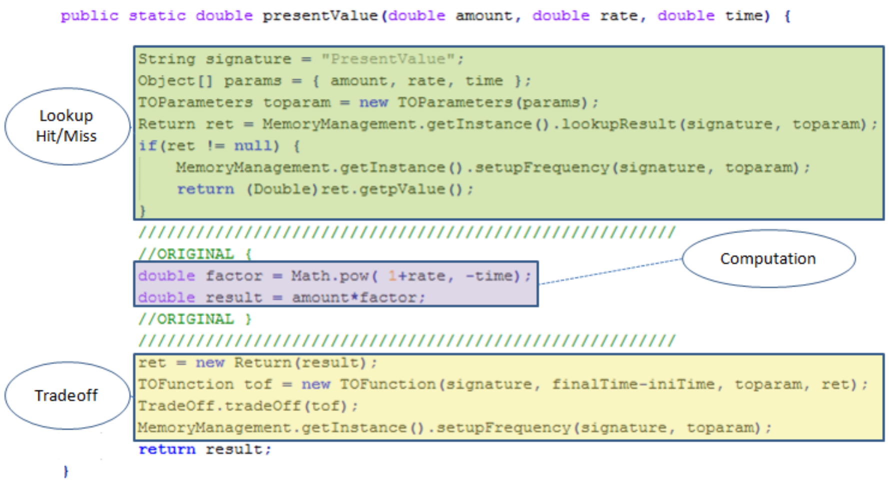


Figure 4.2: Execution Flow and Code Blocks

Furthermore, the correspondence between flow and code is presented in Figure 4.3 where the colors related to each block is relative to the colors in flow presented in Figure 4.2.



```
public static double presentValue(double amount, double rate, double time) {
    String signature = "PresentValue";
    Object[] params = { amount, rate, time };
    TOParameters toparam = new TOParameters(params);
    Return ret = MemoryManagement.getInstance().lookupResult(signature, toparam);
    if(ret != null) {
        MemoryManagement.getInstance().setupFrequency(signature, toparam);
        return (Double)ret.getpValue();
    }
    ////////////////////////////////////////////////////////
    //ORIGINAL {
    double factor = Math.pow( 1+rate, -time);
    double result = amount*factor;
    //ORIGINAL }
    ////////////////////////////////////////////////////////
    ret = new Return(result);
    TOFunction tof = new TOFunction(signature, finalTime-iniTime, toparam, ret);
    TradeOff.tradeOff(tof);
    MemoryManagement.getInstance().setupFrequency(signature, toparam);
    return result;
}
```

Figure 4.3: Source Code related to Flow Blocks

The presented source of Figure 4.3 shows us up the Present Value function operating with Full Tabulation Trade-Off mode explained earlier. That is why there are no Mathematical (average and standard deviation) attributes settled inside the modified source code. In this way, there is no information Trade-Off module can use to discard the values before inserting or not inside the Functions Table structure.

4.2.2 Time Consuming Blocks

Going deeper in analysis of the Flow Diagram presented in Figure 4.2 we can get the division of time consumption:

- Lookup Miss: Time spent to look for a specific value that is not tabulated. The Memory management try to access the HashMaps with desired key but a null return is sent back;

- Computation: Time spent to compute the original function;
- Trade-Off: Time spent by the Trade-Off to decide either to tabulate or not the value;
- Lookup Hit: Time spent to look for a specific tabulated value.

4.2.3 Variables

Besides the time consuming variables, to better describe the model, is used:

- Modified Computation Time: Time spent to compute the modified function;
- Hits Percentage: Percentage of hits provided by the system;
- Break-even Hits Percentage: Percentage of hits needed to achieve the break-even point;
- Time saved: Difference between Computation Time and Modified Computation Time;
- Hits Number: Number of times a point was found tabulated;
- Miss Number: Number of times a point was not found tabulated.

The variables are described in Table 4.1.

4.2.4 Flow Execution Analysis

Introduction

The idea of this subsection is to come up with the minimum hit ratio needed to guarantee the same execution time for the optimized function.

Considering α % of hits, it is possible to write:

$$t_{mod} = \alpha * t_{hit} + (1-\alpha) * (t_{miss} + t_{comp} + t_{tradeoff})$$

$$\Delta_{time} = t_{comp} - t_{mod}$$

If Δ=0 the break-even point is found. So:

$$t_{comp} - t_{mod} = 0 \Rightarrow t_{comp} = t_{mod}$$

Substituting $t_{comp} = t_{mod}$ in the first equation:

$$t_{comp} = \alpha * t_{hit} + (1-\alpha) * (t_{miss} + t_{comp} + t_{tradeoff})$$

$$\alpha_{be} = \frac{t_{miss} + t_{tradeoff}}{t_{miss} + t_{tradeoff} + t_{comp} - t_{hit}}$$

Results

This result can be used to check if either the function can be tabulated or not, because as an assumption α was considered the hit percentage wich means $0 \leq \alpha \leq 1$.

Thus, any function with α out of this range is not worth to tabulate and can be ignored as a part of the system.

Moreover, α_{be} is also the minimum value of hits that the system needs to provide in order to save execution time.

4.2.5 Time Saving Analysis

Introduction

The idea of this subsection is to come up with a superior boundary of time saving for the optimized function.

Considering $\Delta_{time} = t_{comp} - t_{mod}$, it is possible to write:

$$\Delta_{time} = t_{comp} - \alpha * t_{hit} - (1 - \alpha) * (t_{miss} + t_{comp} + t_{tradeoff})$$

As $0 \leq \alpha \leq 1$, using the extreme points of α:

$$\alpha = 0 \Rightarrow \Delta_{time} = -t_{miss} - t_{tradeoff}$$

$$\alpha = 1 \Rightarrow \Delta_{time} = t_{comp} - t_{hit}$$

Results

Hence, the maximum Δ_{time} is given for $\alpha = 1$ and it also gives a superior boundary for the execution time saving.

4.2.6 Conclusion

Supposing processing time as an approach for power consumption, as lower is the processing time as higher will be the power saving.

Bearing that in mind, α can be used to distinguish between the energy saving potential for a given function. Furthermore, the best functions to tabulate are the ones with $\alpha_{be} \rightarrow 0$.

Table 4.1: Alpha Variables Table

Block Name	Variable
Lookup Miss	t_{miss}
Computation	t_{comp}
TradeOff	$t_{tradeoff}$
Lookup Hit	t_{hit}
Modified Computation Time	t_{mod}
Hits Percentage	α
Break-even Hits Percentage	α_{be}
Time saved	Δ_{time}
Hits Number	N_{hits}
Miss Number	N_{miss}

In order to compare different functions, the α for each one is given by:

$$\alpha = \frac{N_{hits}}{N_{miss} + N_{hits}}$$

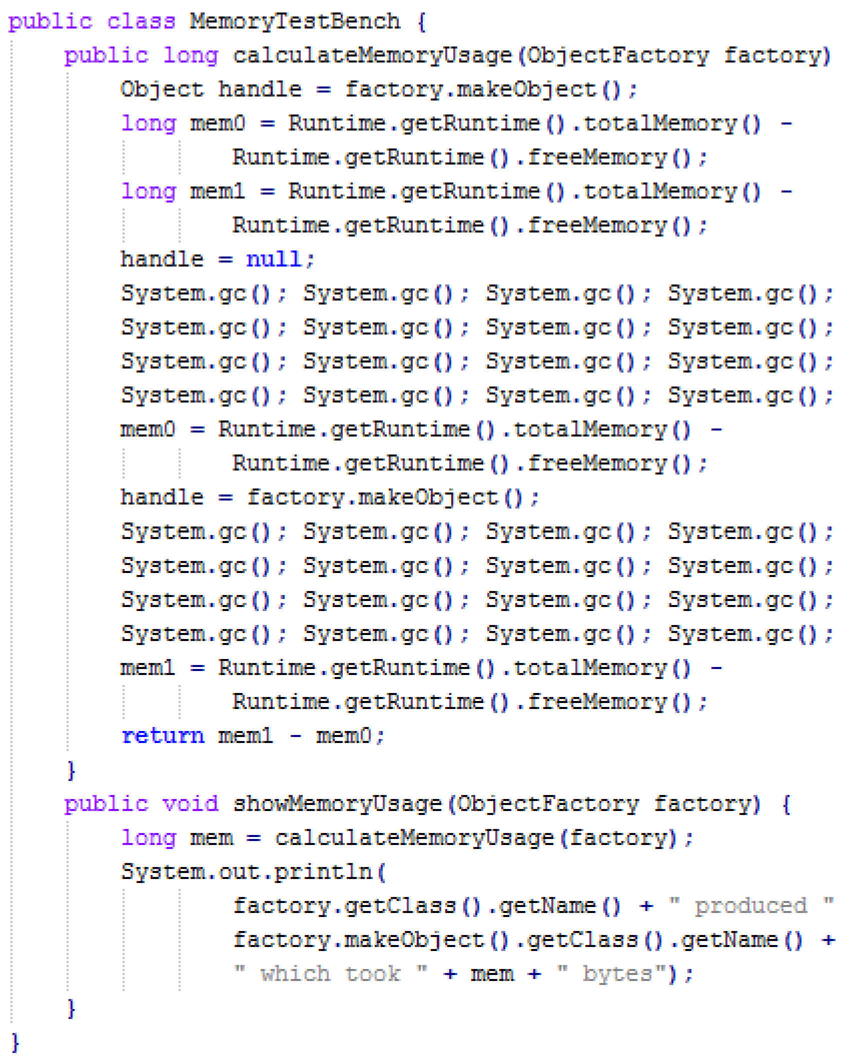
Therefore, having α as far as possible from α_{be} is the best choice for saving energy.

4.3 Allocated Memory Control

One of the issues regarding Java Low-Level Applications is the control of memory allocation by the Objects. Since Java does not provide an API to verify how the memory is consumed by the allocated objects, a set of workarounds are used to estimate the memory that is used by Java. And as Java uses a virtual machine which controls the memory by itself, is not possible to test and to measure the allocation of the objects since it is dynamic and controlled by the JVM.

4.3.1 Metrics Extraction

In this way, memory management is not possible to implement using Java language. For this purpose, usually are used low-level languages such as C and C++. Thus, this work proposes an approach to estimate the memory usage by the objects that are kept in memory to tabulate the inputs and outputs of modified functions.



```java
public class MemoryTestBench {
    public long calculateMemoryUsage(ObjectFactory factory) {
        Object handle = factory.makeObject();
        long mem0 = Runtime.getRuntime().totalMemory() -
                Runtime.getRuntime().freeMemory();
        long mem1 = Runtime.getRuntime().totalMemory() -
                Runtime.getRuntime().freeMemory();
        handle = null;
        System.gc(); System.gc(); System.gc(); System.gc();
        System.gc(); System.gc(); System.gc(); System.gc();
        System.gc(); System.gc(); System.gc(); System.gc();
        System.gc(); System.gc(); System.gc(); System.gc();
        mem0 = Runtime.getRuntime().totalMemory() -
                Runtime.getRuntime().freeMemory();
        handle = factory.makeObject();
        System.gc(); System.gc(); System.gc(); System.gc();
        System.gc(); System.gc(); System.gc(); System.gc();
        System.gc(); System.gc(); System.gc(); System.gc();
        System.gc(); System.gc(); System.gc(); System.gc();
        mem1 = Runtime.getRuntime().totalMemory() -
                Runtime.getRuntime().freeMemory();
        return mem1 - mem0;
    }
    public void showMemoryUsage(ObjectFactory factory) {
        long mem = calculateMemoryUsage(factory);
        System.out.println(
                factory.getClass().getName() + " produced " +
                factory.makeObject().getClass().getName() +
                " which took " + mem + " bytes");
    }
}
```

Figure 4.4: MemoryTestBench Used to Compute Memory Consumption

The image 4.4 shows a class used to compute how the instantiation of an object consume the memory and allows us to estimate the consume of, for example, the `HashMap` entry overhead, class attributes used to point objects and also the overhead of an object array creation. Since the Java memory fluctuates in the way it allocates and deallocates objects, then, the `MemoryTestBench` presented calls Garbage Collector to stabilize the memory before measuring it by the difference between total and free memory.

4.3.2 Memory Estimation

The Figure 4.5 shows the schema of how objects are positioned among `TOFunctions` structures. Based on that image, we are going to describe the memory usage estimation. First, the letters A, B, C and D points to structures which will handle objects kept in memory and also represents an overhead in Java's memory consumption. Beside the explanation of how the memory is estimated we will refer also to data we have obtained by the use of the `MemoryTestBench` described in the section above.

Therefore, going to estimation information, we enumerate below the information in details:

- *48 Bytes* (HashMap Entry Overhead - A): the memory consumed by the operation of inserting one element inside the `HashMap`. This consume regards reference to hash key, hash value, the object which key refers to and also the 24 bytes used by a empty object;

- *24 Bytes* (TOParameters Object): a simple object in Java costs this bytes number;

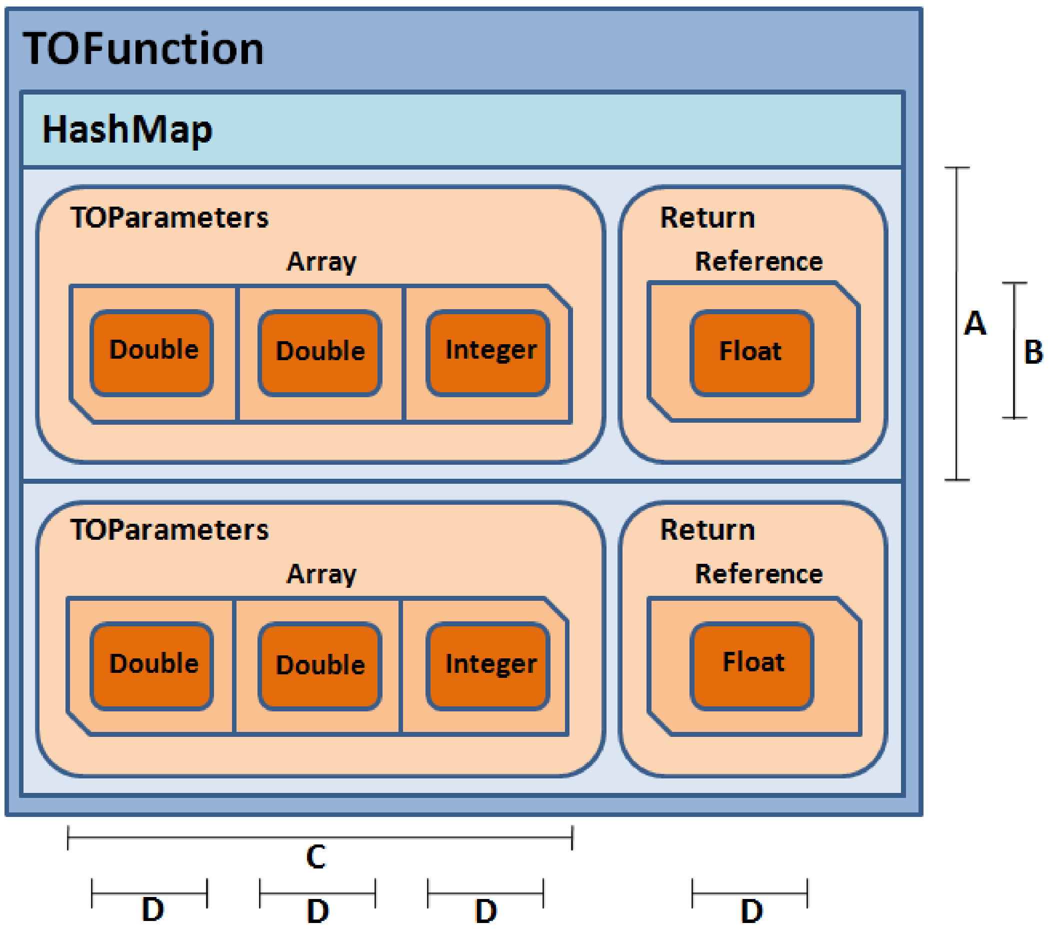


Figure 4.5: Objects organization with labels to be referred in each memory consumption description, enumerated below

- *16 Bytes* (TOParameters Attributes): one reference to `Integer` and `Array` costs 8 Bytes each;
- *24 Bytes* (Array Object Instance - C): one intantiated array costs 24 Bytes. This must be summed to the Array reference that uses 8 Bytes (mentioned in item above);
- *8 Bytes* (Each Array Position - D): since the array position is a reference, it consumes 8 Bytes as any reference;

- *24 Bytes* (Return Object - B): a simple Object in Java, similar to `TOParameters`;
- *16 Bytes* (Return Attributes - B): this amount is summed to the whole cost;
- *32 Bytes* (Data Object - D): this is the cost of each object pointed by array position references individually. This object keeps the input or output value of a function and therefore is a value like `Double`, `Integer`, `Float` and so on. All these objects consume 32 Bytes.

From the description enumerate before, we reach the calculation that provides the memory consumption of each entry inside the `TOFunction HashMap`. Since the functions will increase their values accordingly to the system execution, the HashMap will become the uniquely important memory measurement. `TOFunction` and other objects overhead in comparison become neglected as the memory required does not get over than a couple of bytes. So these last values will not be taken into account for the memory usage control.

The formula is: (24 + 8 + 8 + 24 + <Parameters_Array_Size> * (8 + 32) + 24 + 16 + 32 + 48) to get the `TOFunction` entry size.

Chapter 5

Tests and Results

In this chapter is presented the tests used to evaluate all savings obtained by introducing the software usage by financial functions. First it was analyzed many financial applications that could be used by this kind of approach. After the system was stable it was possible to identify some applications worth applying this method, so there were two main applications being tested and used to identify how the parameters could be tuned in order to achieve the best behavior and maximum savings.

5.1 Experimental Setup

Regarding a specific application, for each of the two functions previously chosen a test class was implemented individually and for a function call it has been ran ten times the same function with different inputs, this methodology was used in order to make all tests more consistent and faster, because for every call it was needed to come up with random inputs and running ten times in a row makes the time execution faster. For instance, if you execute 10 calls it is going to run 100 times the function.

Moreover, to simulate the execution of the functions as close as possible to the real world, each one of the functions was randomly chosen to execute its iterations. The probability density for the inputs was chosen as Gaussian, based on the central limit theorem, and to select which function to execute was used equally likely probability density.

Furthermore, a precision was given for each input in order to limit the total possible combinations inside the range of two standard deviations chosen by tabulating all the points. This assumption was made to come close to the real workload, for example the stocks listed in most of the stock exchange markets have as minimum movement one cent of the currency.

Another point related to simulation is the environment used to run all the testing routines. It was not used any dedicated environment but personal computer. The station used is a Core 2 Duo 2.1 Ghz with 3GB of memory. It is running a Windows 7 x64. Since the computer is personal and runs other tasks in paralel, this configuration may create a considerable deviation in simulation times. We have included standard deviation in our results to present the behavior of time fluctuation when the tests were executed.

It is possible to check an example of the tests code below:

Program 1 Example of Test Class.

```
public static void testExecutionsVar() {
    for (nExec = 100000; nExec < nExecTotal; nExec++)
       {
        int testsNumber = nExec;
        int[] fNumber = new int[testsNumber];
        fNumber = MathFunctions.generateRandomFunction
        (2, testsNumber);
        for (int i = 0; i < testsNumber; i++) {
            switch (fNumber[i]) {
                case 0:
                    testbinomialOptionPricing();
                    break;
                case 1:
                    testimpliedVolatility();
                    break;
            }
        }
        MemoryManagement.resetMemory();
    }
 }
```

Program 2 Example of Test Function Class.

```
public static void testbinomialOptionPricing() {
        Integer inputsNumber = 10;
        Integer nPeriods = 1000;
        double[] S = MathFunctions.generateRandom
        (100, 0, inputsNumber, 2);
        double[] X = MathFunctions.generateRandom
        (120, 0, inputsNumber, 2);
        double[] T = MathFunctions.generateRandom
        (1, 0, inputsNumber, 2);
        double[] r = MathFunctions.generateRandom
        (0.05, 0.01, inputsNumber, 3);
        double[] v = MathFunctions.generateRandom
        (0.5, 0.1, inputsNumber, 2);
        for (int i = 0; i < inputsNumber; i++) {
            ModifiedFunctions.binomialOptionPricing
            (S[i], X[i], T[i], r[i], v[i], nPeriods);
        }
    }
```

5.2 Benchmark Applications

The two functions chosen to run the tests were Binomial Option Pricing and Implied Volatility, in order to choose these specific functions it was used the approach explained in Section 4.2. All the financial functions listed in Section 2.3 were modified and tested, using the alpha approach as showed in the Table 5.1. After all these tests and also using the fact that $0 \leq \alpha \leq 1$, as explained in Section 4.2.4, it was clear which functions were worth tabulating.

Calculating the alpha for a specific function gives important information about the behavior of this function when applied the modifications and integrated to the system.

Table 5.1: Functions Alpha Values

Function Name	$\alpha(\%)$
ROI	179,24
FV	182,41
PV	122,72
NPV	155,82
XIRR Time	22,92
BOPM	3,19
Implied Volatility	4,59
Duration	158,67
Convexity	149,64

To exemplify the process of function selection it is presented this flowchart below:

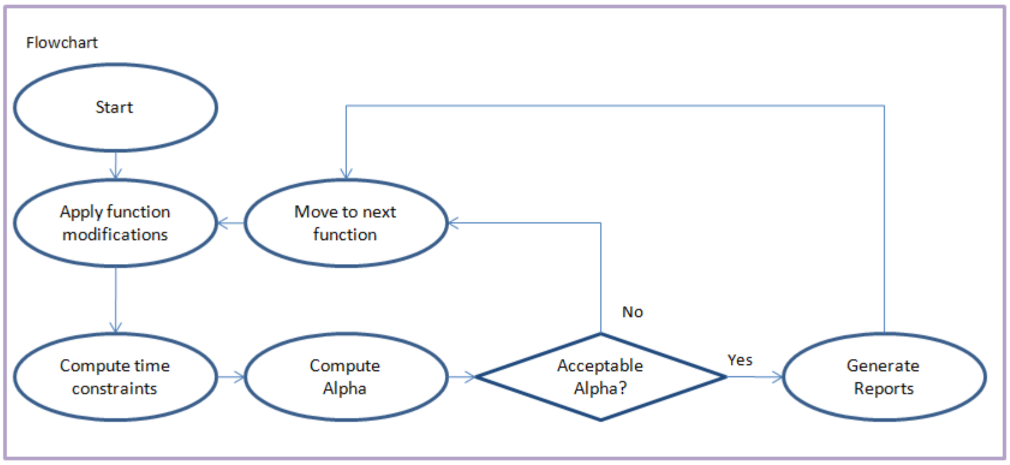


Figure 5.1: Choosing Functions

Where:

- Apply function modifications: It means apply the modifications proposed in Subsection 3.5.3
- Compute time constraints: Compute the time constraints needed to calculate the function Alpha, this approach is proposed in Section 4.2
- Compute Alpha: Compute Alpha itself, given by:

$$\alpha_{be} = \frac{t_{miss} + t_{tradeoff}}{t_{miss} + t_{tradeoff} + t_{comp} - t_{hit}}$$

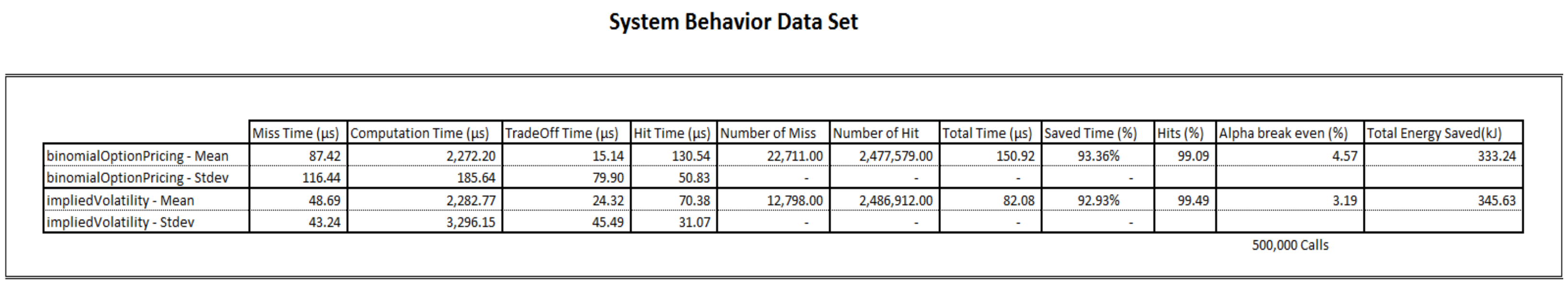
- Generate Reports: Using Alpha computed on the last step, it is possible to classify a function, regarding memory size and also Energy Savings.

5.3 Results

Concerning the possible system speed up and energy saving, it is necessary to have a better and deeper knowledge about its behavior. In this interest it was built two different sets of tests, the first one was made using the total amount of memory available on the computer and the second set was simulated by varying the memory size to identify important points, as minimum memory size or also maximum number of hits.

5.3.1 First Test

For this test was used the total available memory of the java virtual machine (1 GB), moreover it was done by executing 500,000 calls which means approximately 2,500,000 executions of each function. The total energy saved was computed by using as assumption the power value published by Fiorelli and Polleto thesis [7] for this specific computer, the value is 62,83 W for fully stressed workload ($P_{consumption}$).



System Behavior Data Set

	Miss Time (µs)	Computation Time (µs)	TradeOff Time (µs)	Hit Time (µs)	Number of Miss	Number of Hit	Total Time (µs)	Saved Time (%)	Hits (%)	Alpha break even (%)	Total Energy Saved(kJ)
binomialOptionPricing - Mean	87.42	2,272.20	15.14	130.54	22,711.00	2,477,579.00	150.92	93.36%	99.09	4.57	333.24
binomialOptionPricing - Stdev	116.44	185.64	79.90	50.83	-	-	-	-			
impliedVolatility - Mean	48.69	2,282.77	24.32	70.38	12,798.00	2,486,912.00	82.08	92.93%	99.49	3.19	345.63
impliedVolatility - Stdev	43.24	3,296.15	45.49	31.07	-	-	-	-			

500,000 Calls

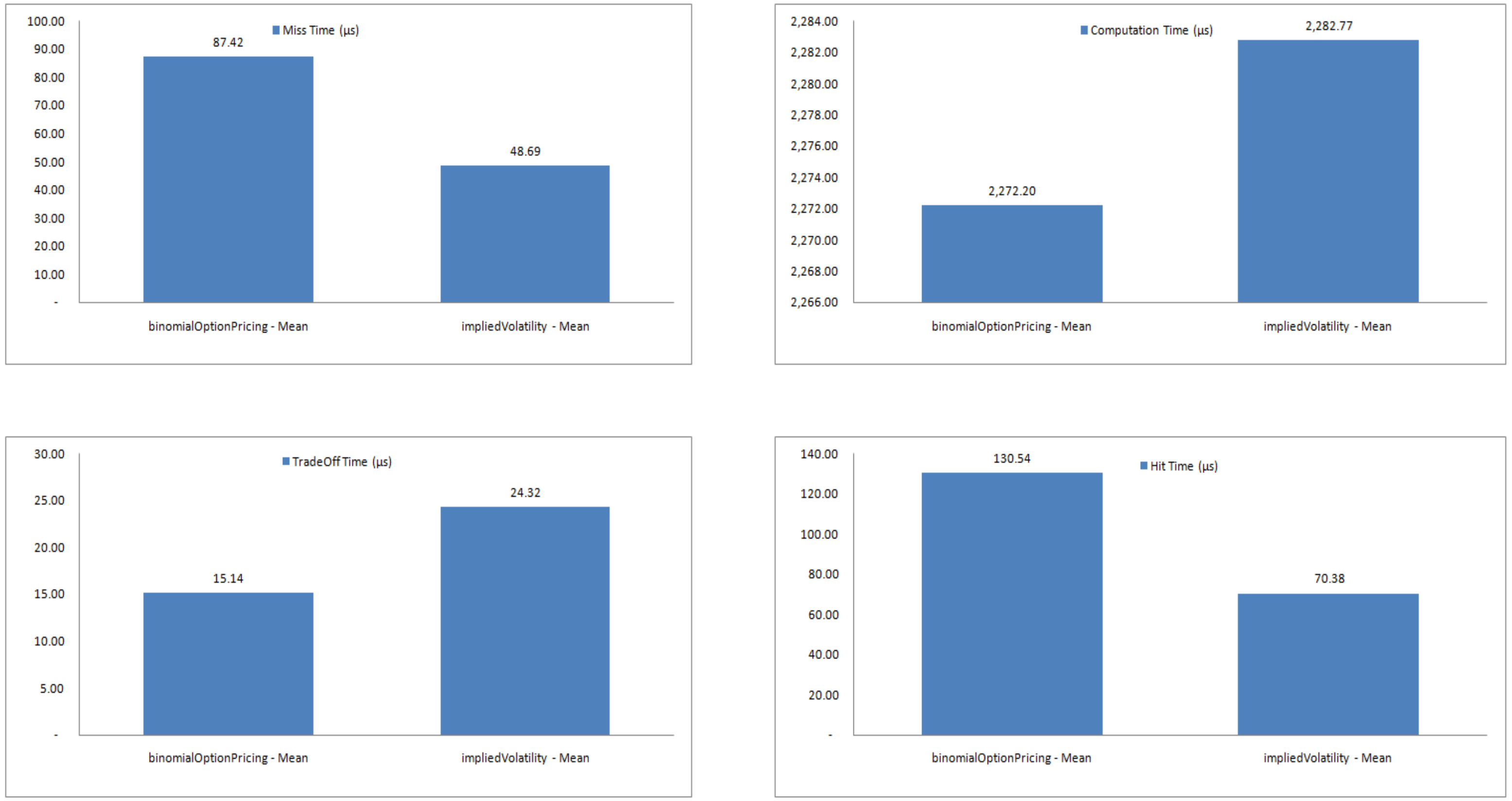


Figure 5.2: First Test

Where the fields on the Report represented by Figure 5.2 are described as:

- Miss Time (t_{miss}): Time spent to look for a specific value that is not tabulated
- Computation Time (t_{comp}): Time spent to compute the original function
- TradeOff Time ($t_{tradeoff}$): Time spent by the trade off to decide either to tabulate or not the value
- Hit Time (t_{hit}): Time spent to look for a specific tabulated value
- Number of Miss (N_{miss}): Number of times a point was not found tabulated
- Number of Hit (N_{hits}): Number of times a point was found tabulated
- Hits (α): Percentage of times a point was found tabulated.Computed as:

$$\alpha = \frac{N_{hits}}{N_{hits} + N_{miss}}$$

- Total Time (t_{total}): Number representing the new computation time using the system, given by :

$$t_{total} = \alpha * t_{hit} + (1 - \alpha) * (t_{miss} + t_{comp} + t_{tradeoff})$$

- Saved Time (t_{saved}): Difference between Computation Time and Modified Computation Time in percentage, computed as:

$$t_{saved} = \frac{t_{comp} - t_{total}}{t_{comp}} * 100\%$$

- Alpha Breakeven (α_{be}): Minimum percentage of hits needed to achieve the break-even point.

$$\alpha_{be} = \frac{t_{miss} + t_{tradeoff}}{t_{miss} + t_{tradeoff} + t_{comp} - t_{hit}}$$

- Total Energy Saved (E_{saved}): Energy saved in kJ, give by:

$$E_{saved} = (t_{comp} - t_{total}) * P_{consumption} * (N_{hits} + N_{miss})$$

5.3.2 Analysis

It is possible to conclude that the tests were successful, because the execution time of both functions was reduced. In other words, for the same workload set, it was needed to use less processing time after applying the optimization. This can be translated to energy saving by using the *saved time*, given by:

$$t_{saved} = \frac{t_{comp} - t_{total}}{t_{comp}} * 100\%$$

Regarding the speed up, it is computed that instead of running for 190 minutes the same work was done in 10 minutes. Translating to energy, it was saved 680kJ in 10 minutes of execution.

5.3.3 Second Test

The impact of memory availability on optimization. To this end, memory size was change in a range from 500 bytes to 100MB and the number of calls was set to 10,000 calls.

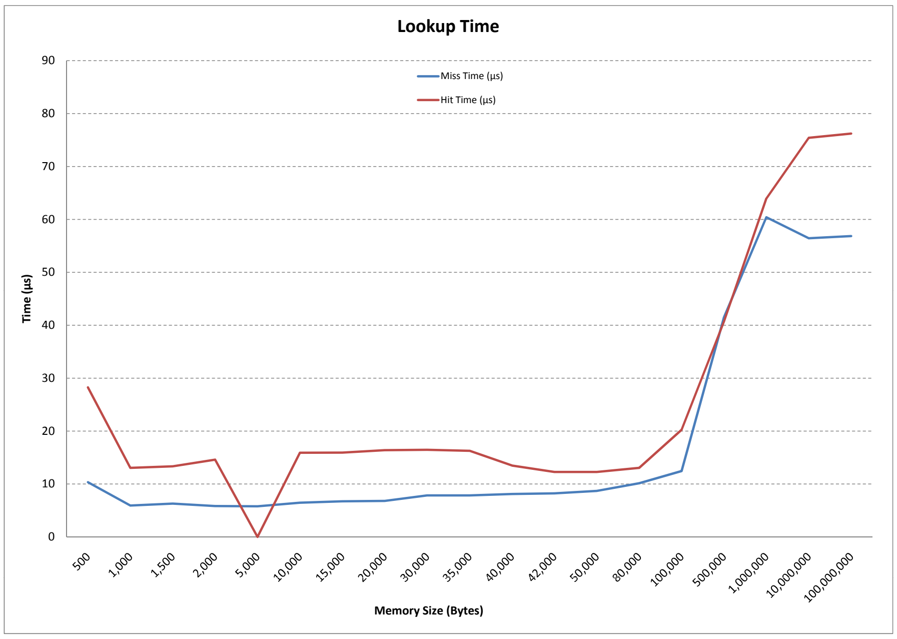


Figure 5.3: Lookup Time for the Binomial Option Pricing Function

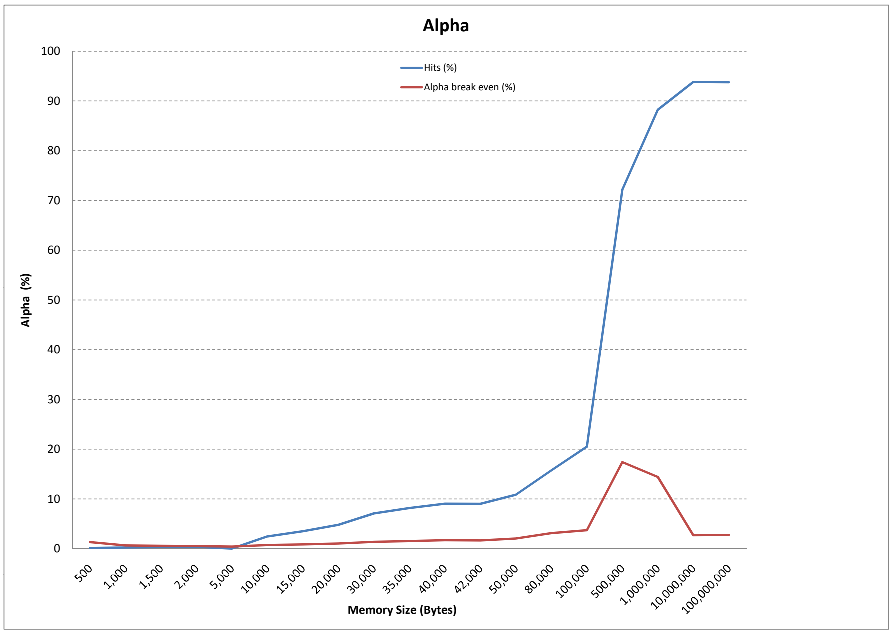


Figure 5.4: Percentage of hits (α) and the alpha breakeven (α_{be}) for the Binomial Option Pricing Function

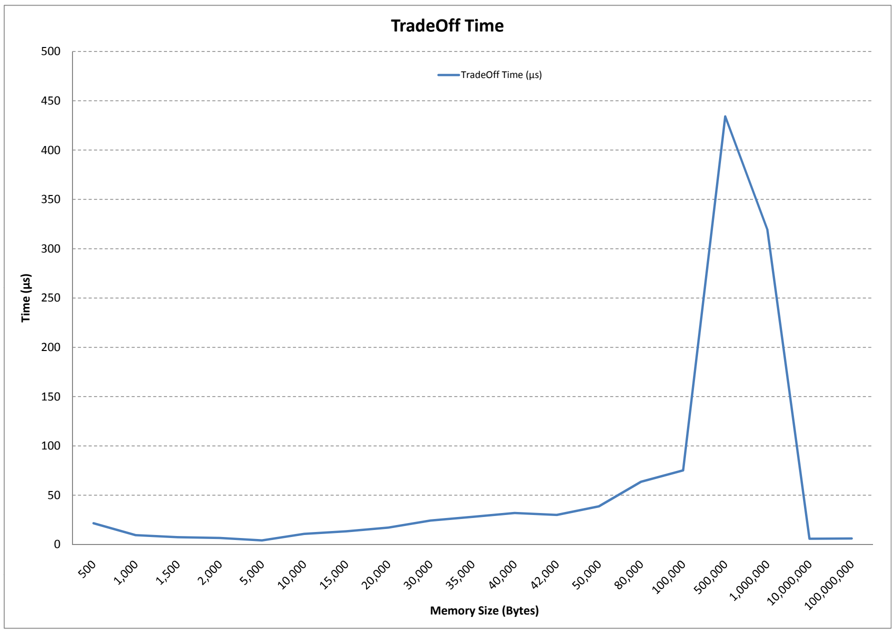


Figure 5.5: TradeOff Time for the Binomial Option Pricing Function

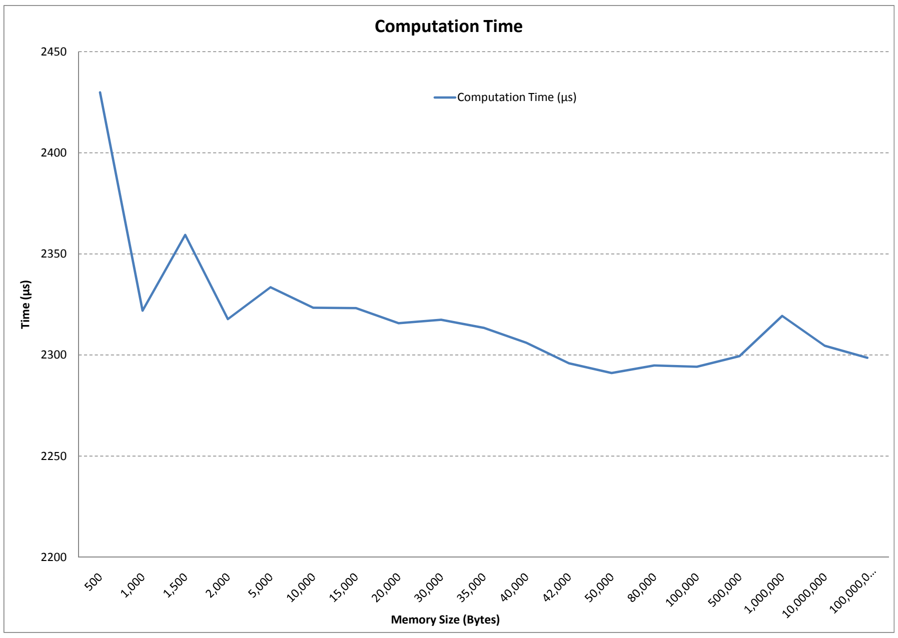


Figure 5.6: Computation Time for the Binomial Option Pricing Function, considering the function withou any optimization

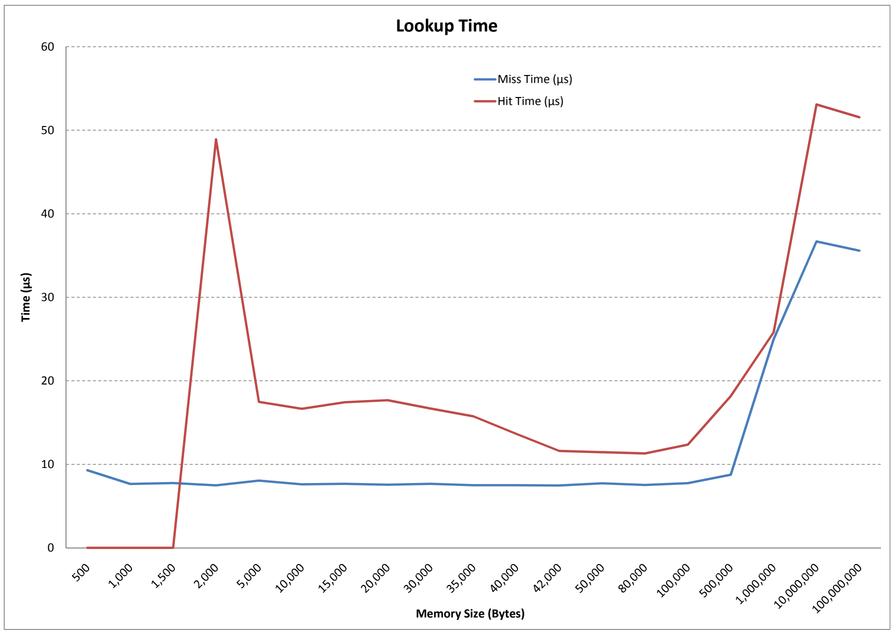


Figure 5.7: Lookup Time for the Implied Volatility Function

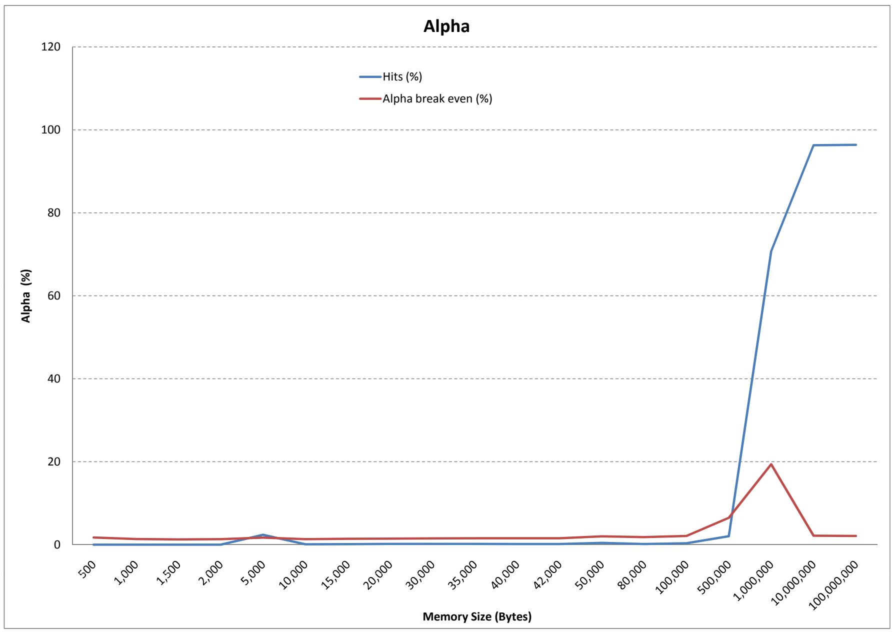


Figure 5.8: Percentage of hits (α) and the alpha breakeven (α_{be}) for the Implied Volatility Function

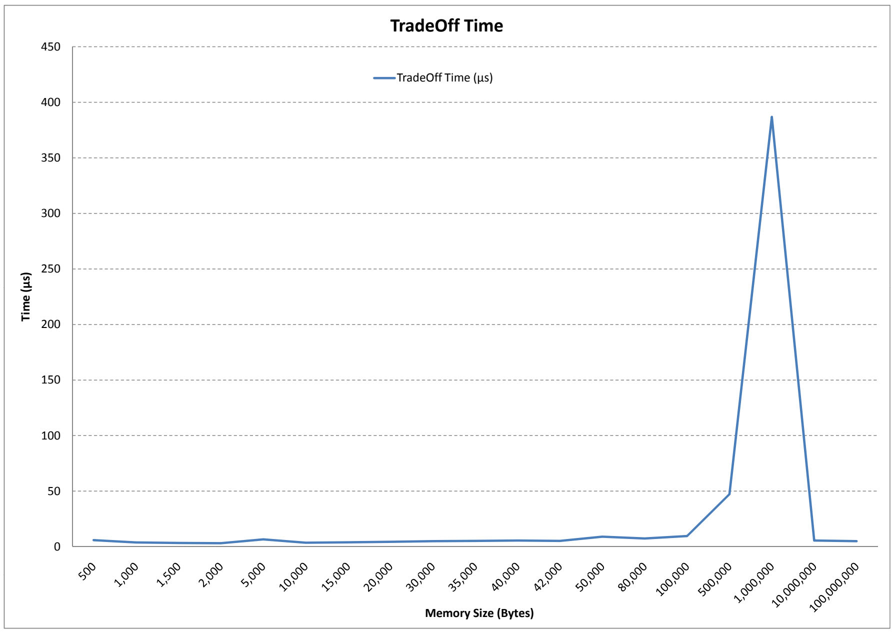


Figure 5.9: TradeOff Time for the Implied Volatility Function

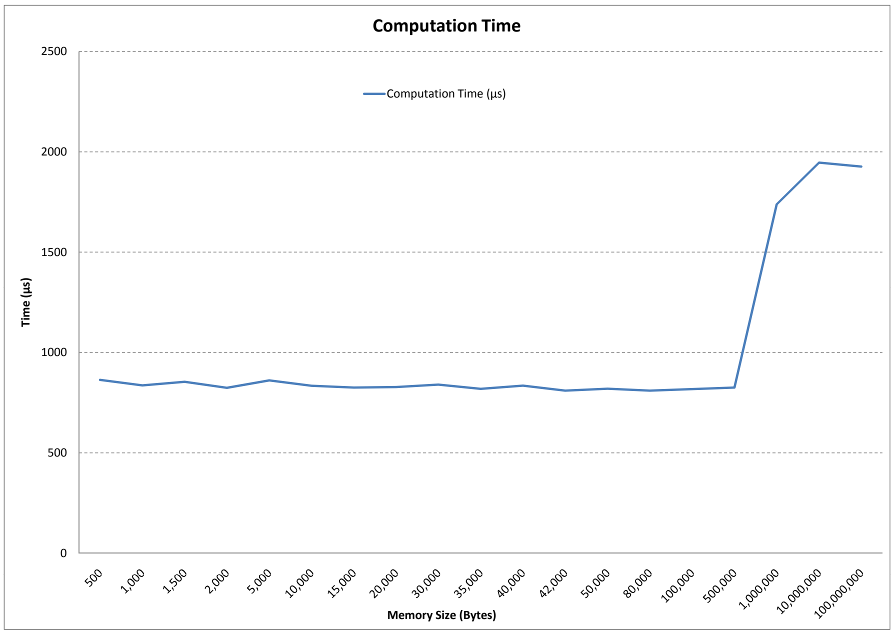


Figure 5.10: Computation Time for the Implied Volatility Function, considering the function withou any optimization

5.3.4 Analysis

Observing the Alpha Breakeven Figures: 5.4, 5.8 and the Trade-Off Time Figures: 5.5, 5.9 is possible to notice they have a correlation, based on its shapes and also the maximum point of each one stands on the same memory size value. This correlation exists due to Alpha Breakeven computation explained in Section 4.2. The trend reversal presented can be explained by the total number of combinations given the input parameters. Additionally, the hits rate becomes stable after the maximum alpha point and it is also explained by the tabulation of a high percentage of the possible inputs.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Green IT is a research field becoming even more important nowadays when natural resources are considered important to sustainable development. With importance increasing and researches, different approaches start being created in the direction of natural resources saving to avoid environmental impact. Since energy is directly related to Green impact, and beside is also becoming an expensive attribute in overall companies budget, the Green IT is trying to find solutions not intrusive to already existent solutions. Why?! Because modifying equipments and change systems running currently in industry organizations are very expensive and may not guarantee the reliability and safety previously analyzed.

Then, taking into account companies with huge datacenters to provide results and processing of their business, this work started focusing in software layer which is an very customizable field and also flexible to changes. Earlier researches had shown us up that the way the software layer is executed in huge datacenters with high data load impact differently in the overall hardware behavior by means of microprocessor power consumption, number of memory banks powered permanently and also the heat generated. Heat, a very important issue in Green IT and datacenters configuration, handles a large slice of the power consumption in IT since the air conditioners usually are high consumer equipments.

This work aimed to create mechanisms to reduce the microprocessor energy consumption by decreasing work load in chips. The main idea is to take in consideration methods able to be modified from the original behavior trying to substitute the original code by one responsible by tabulating the data usually reprocessed. Those methods, known also as Pure Functions already mentioned in this work are methods with only mathematical code, without any other type or interaction with, for instance, graphical user interface, network, or anything else.

Regarding software modification, this work is also part of a large architecture with Pure Function recognition, Trade-Off Module, Memory Management and also Business Intelligence to analyze generated data. More precisely, this thesis concerns the Trade-Off Module and Memory Management modules, with the idea to decide whether some information will need to be kept in memory of not also managing the remaining space and also function priorities already described in previous sections of this document. The development of all modules were done in java and was exported a set of different statistics about systems behavior in different usage conditions.

The results got in systems analysis overcome our expectations with high indexes of time and energy saving. Indeed, the idea of tabulate all the complex mathematical inputs and output from the Pure Functions have dealt with energy saving by decreasing of 100% workload time on microprocessor. The percentages run around 90% of saving considering large amount of data being tabulated and also the approaches created to measure numbers of hits and misses in memory architecture to keep information. But the actual impact of this huge saving is approximately 20%, taking into consideration the benchmark used, in the end just two functions out of nine were worth applying the optimization, which tells us that 22% of the applications chosen had around 90% of savings.

Therefore, handling the software layer in Green IT is an important issue to be researched since saving microprocessor workload will reduce computation time allowing the hardware platform to hold more workload. Will also reduce the heat generated by the architecture making air conditioners decrease the energy required to cool the environment. Considering this cascade the overall power consumption of whole Data Center structure will be cut down.

6.2 Future Work

Considering promising results presented in this work, there is a set of enhancements planned to be put in practice in subsequent work steps. The first future work to be developed is the improvement of the deletion mechanism of the Trade-Off module. Nowadays, the deletion behavior iterates through the ordered by priorities data structure to eliminated that entries with priorities lower than the new entry to be inserted in Memory. The call to deletion process is made passively, in other words, there must be a call to Trade-Off module to verify if the entry must or not be kept in cache system and if must be, then space must be find to new data. That is when the deletion operates. Also, there is a configuration to define the number of calls required to execute the cleaning methods to reduce performance impact of Trade-Off functionality. Thus, the improvement would be the creation of an active method to delete useless or low priorities entries from memory without the need to call the method by the original system. This will also ease the control of memory percentage available to each function tabulated.

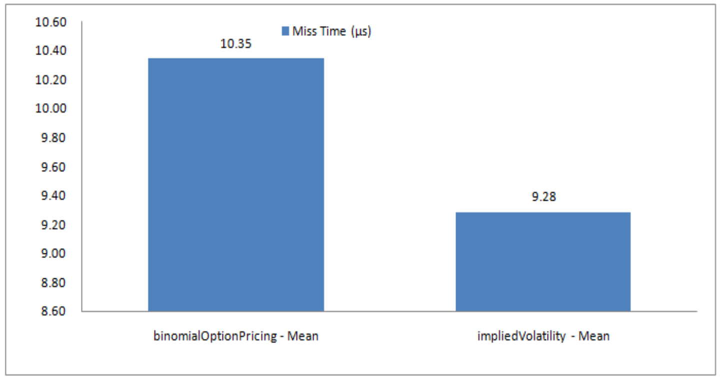
Beside the approach created in this work to define whether a function must be tabulated of not considering execution times, a possible enhancement is the insertion of new attributes for this verification. The idea would be the insertion of a Beta correction factor which would correlate the pure function times (original execution time and new modified function times: hit/miss, computation and trading-off), power consumption considering idle/busy microprocessor and memory powering - constant power required to each inserted memory bank. A more complex set of variables to analyze the tabulation or not for a function would give more precise information about functions behavior relating to memory usage.

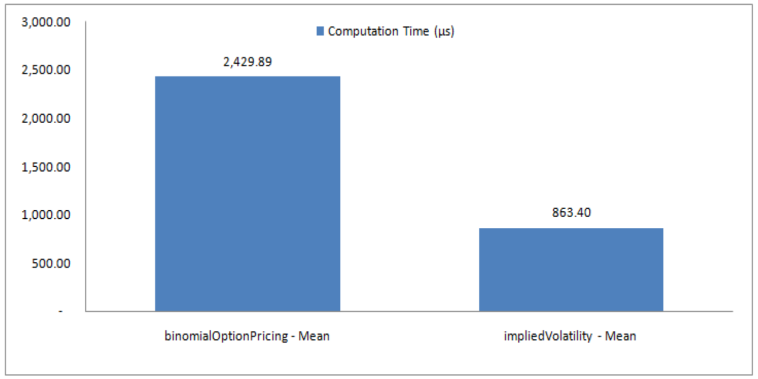
Appendix A- Simulation Reports

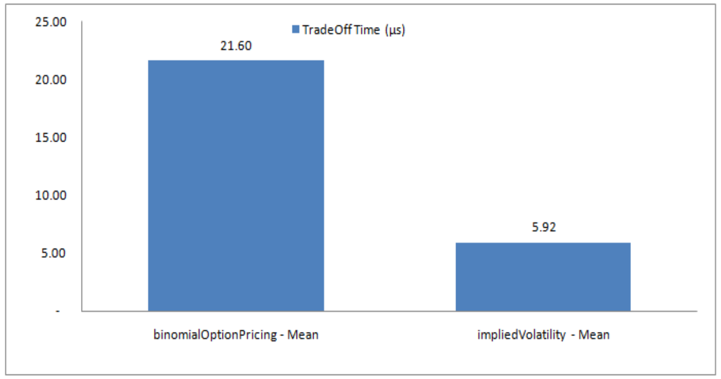
In this appendix is going to be presented the reports generated for some of the points plotted on Section 5.3.3.

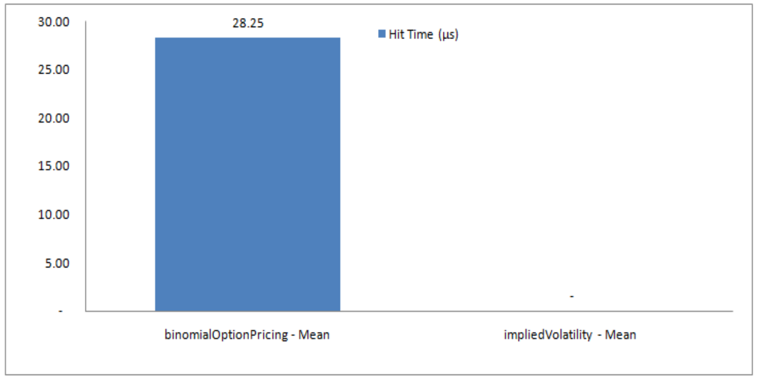
System Behavior Data Set - 500 Bytes

	Miss Time (µs)	Computation Time (µs)	TradeOff Time (µs)	Hit Time (µs)	Number of Miss	Number of Hit	Total Time (µs)	Saved Time (%)	Hits (%)	Alpha break even (%)	Total Energy Saved(kJ)
binomialOptionPricing - Mean	10.35	2,429.89	21.60	28.25	49,651.00	69.00	2,458.46	-1.18%	0.14	1.31	(0.09)
binomialOptionPricing - Stdev	108.30	707.60	1,446.44	30.85	-	-	-	-			
impliedVolatility - Mean	9.28	863.40	5.92	-	50,280.00	-	878.60	-1.76%	-	1.73	(0.05)
impliedVolatility - Stdev	80.87	1,840.17	66.48	-	-	-	-	-			
										10,000 Calls	500 Bytes





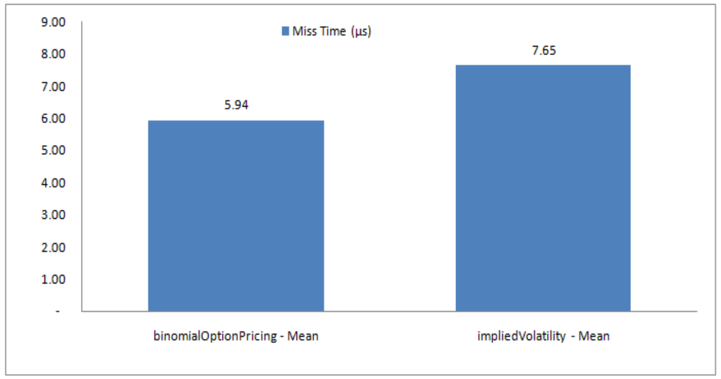


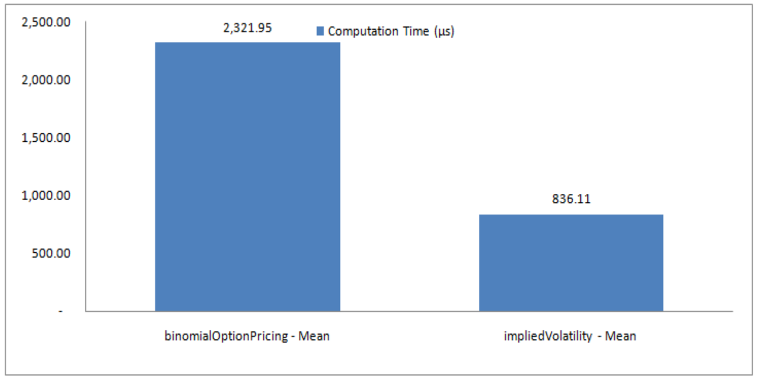


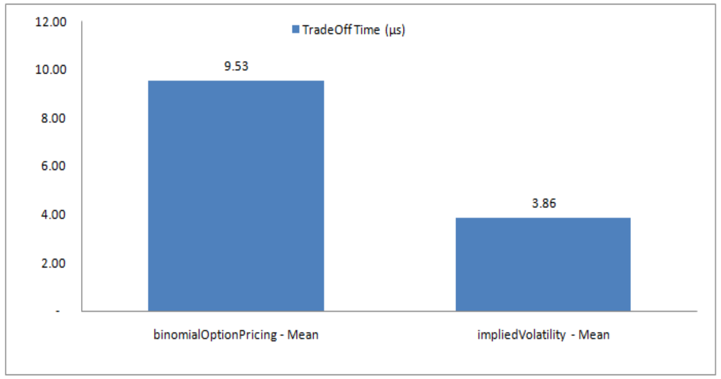
System Behavior Data Set - 1k Bytes

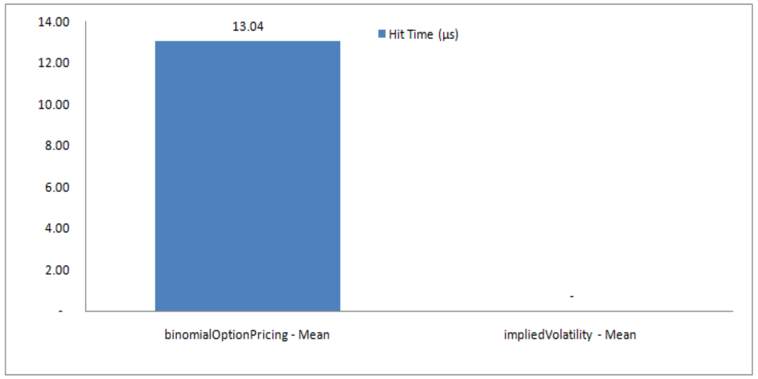
	Miss Time (µs)	Computation Time (µs)	TradeOff Time (µs)	Hit Time (µs)	Number of Miss	Number of Hit	Total Time (µs)	Saved Time (%)	Hits (%)	Alpha break even (%)	Total Energy Saved(kJ)
binomialOptionPricing - Mean	5.94	2,321.95	9.53	13.04	49,958.00	122.00	2,331.76	-0.42%	0.24	0.67	(0.03)
binomialOptionPricing - Stdev	7.40	137.14	12.68	6.95	-	-	-	-			
impliedVolatility - Mean	7.65	836.11	3.86	-	49,920.00	-	847.62	-1.38%	-	1.36	(0.04)
impliedVolatility - Stdev	6.41	1,727.48	11.41	-	-	-	-	-			

10,000 Calls 1k Bytes





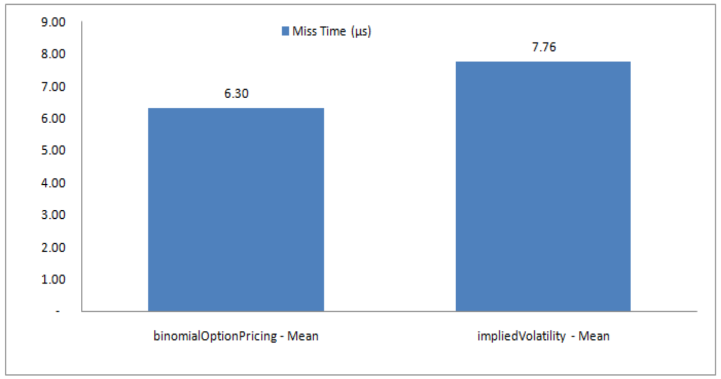


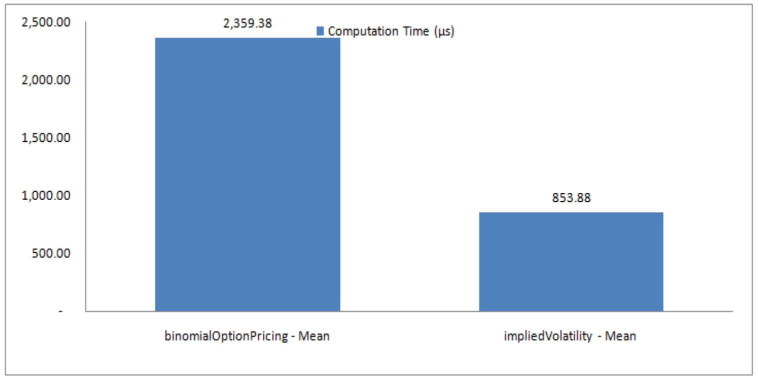


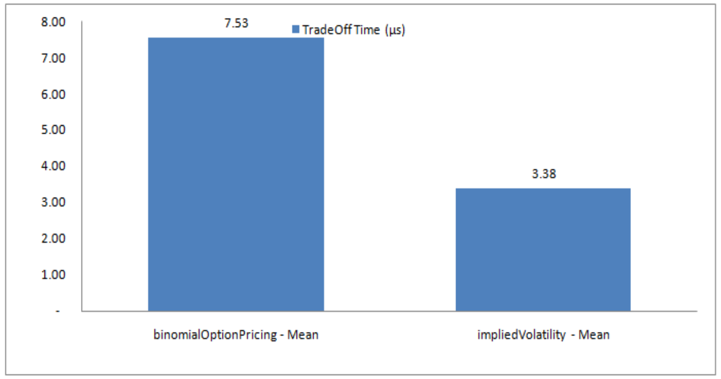
System Behavior Data Set - 1,5k Bytes

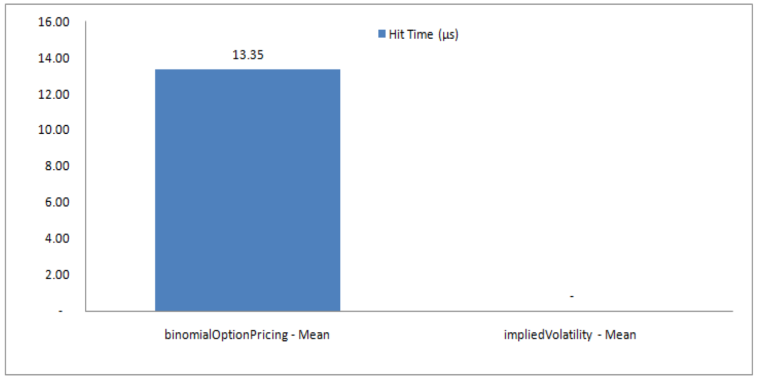
	Miss Time (µs)	Computation Time (µs)	TradeOff Time (µs)	Hit Time (µs)	Number of Miss	Number of Hit	Total Time (µs)	Saved Time (%)	Hits (%)	Alpha break even (%)	Total Energy Saved(kJ)
binomialOptionPricing - Mean	6.30	2,359.38	7.53	13.35	50,257.00	163.00	2,365.58	-0.26%	0.32	0.59	(0.02)
binomialOptionPricing - Stdev	25.37	502.76	57.69	12.72	-	-	-	-			
impliedVolatility - Mean	7.76	853.88	3.38	-	49,580.00	-	865.01	-1.30%	-	1.29	(0.03)
impliedVolatility - Stdev	9.01	1,765.97	6.13	-	-	-	-	-			

10,000 Calls 1,5k Bytes





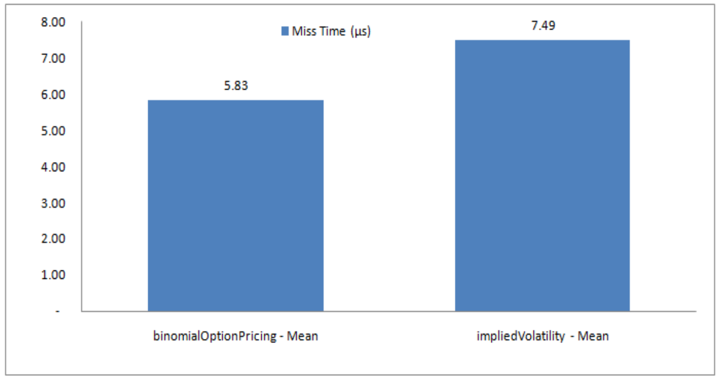


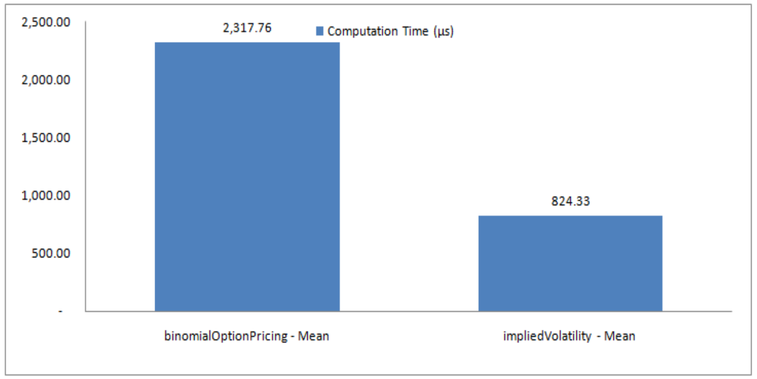


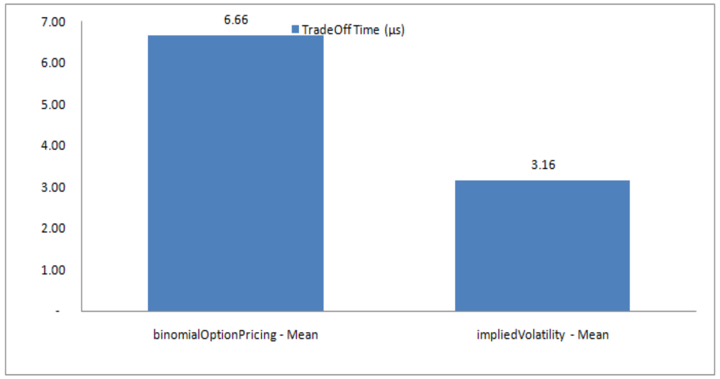
System Behavior Data Set - 2k Bytes

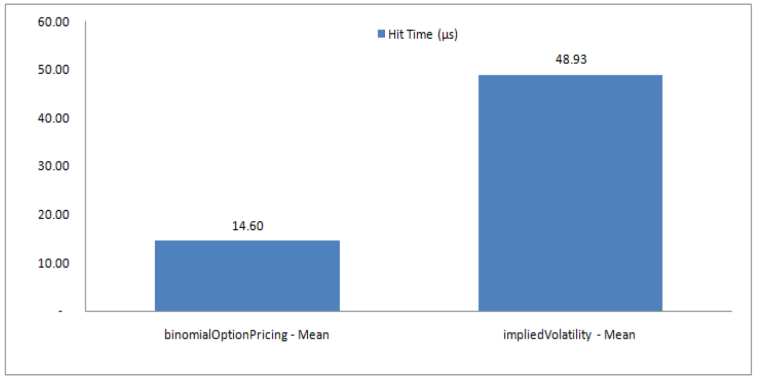
	Miss Time (µs)	Computation Time (µs)	TradeOff Time (µs)	Hit Time (µs)	Number of Miss	Number of Hit	Total Time (µs)	Saved Time (%)	Hits (%)	Alpha break even (%)	Total Energy Saved(kJ)
binomialOptionPricing - Mean	5.83	2,317.76	6.66	14.60	50,126.00	204.00	2,320.87	-0.13%	0.41	0.54	(0.01)
binomialOptionPricing - Stdev	6.43	142.43	7.30	6.93	-	-	-	-			
impliedVolatility - Mean	7.49	824.33	3.16	48.93	49,667.00	3.00	834.93	-1.29%	0.01	1.35	(0.03)
impliedVolatility - Stdev	6.15	1,721.05	5.51	51.86	-	-	-	-			

10,000 Calls 2k Bytes





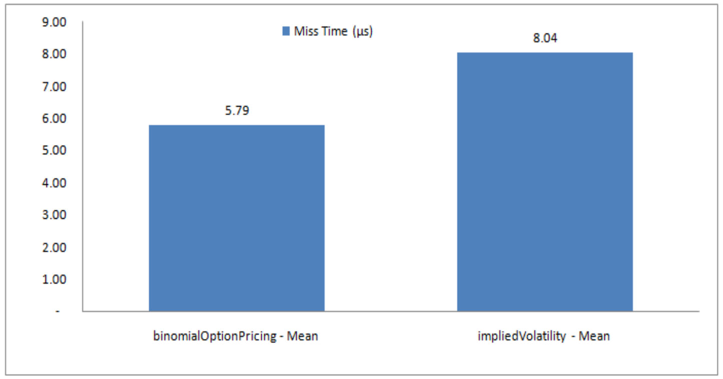


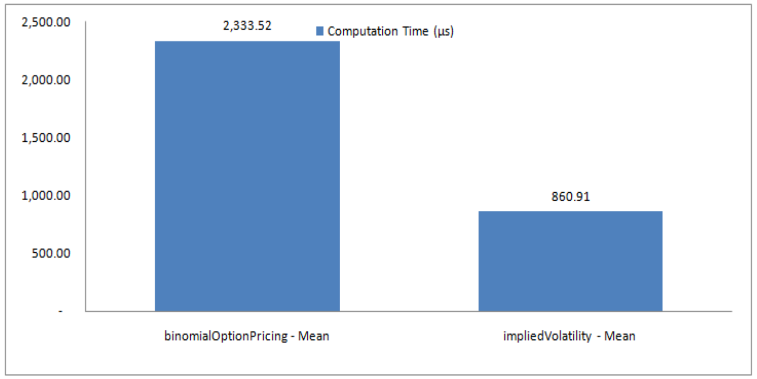


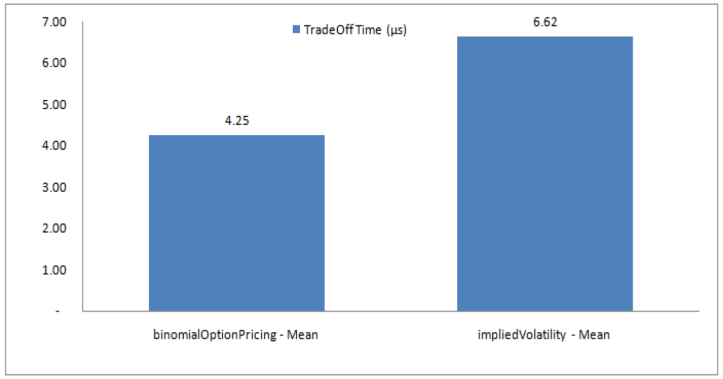
System Behavior Data Set - 5k Bytes

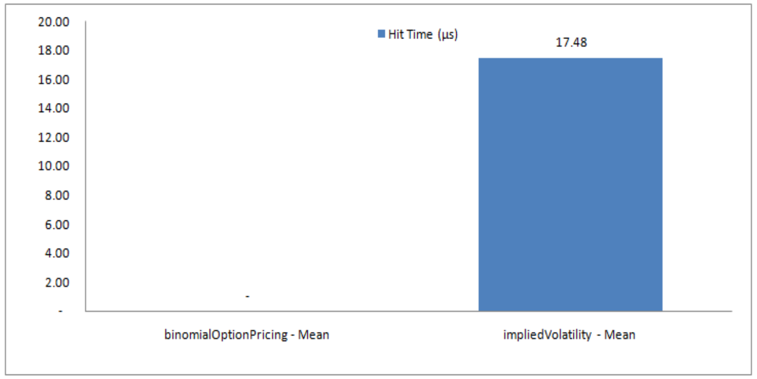
	Miss Time (µs)	Computation Time (µs)	TradeOff Time (µs)	Hit Time (µs)	Number of Miss	Number of Hit	Total Time (µs)	Saved Time (%)	Hits (%)	Alpha break even (%)	Total Energy Saved(kJ)
binomialOptionPricing - Mean	5.79	2,333.52	4.25	-	50,140.00	-	2,343.56	-0.43%	-	0.43	(0.03)
binomialOptionPricing - Stdev	6.20	244.77	8.39	-	-	-	-	-			
impliedVolatility - Mean	8.04	860.91	6.62	17.48	48,666.00	1,194.00	855.02	0.68%	2.39	1.71	0.02
impliedVolatility - Stdev	11.41	1,787.71	7.35	12.45	-	-	-	-			

10,000 Calls 5k Bytes





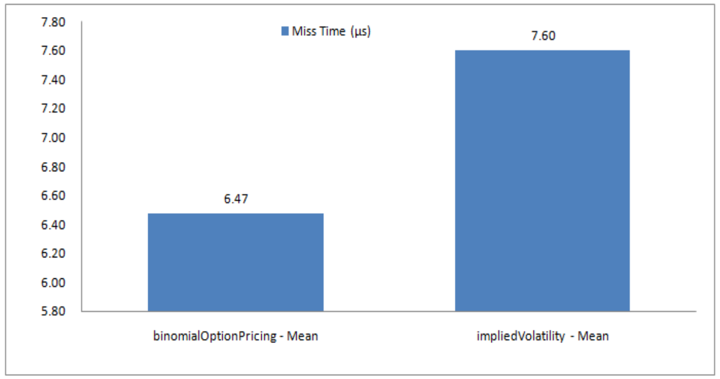


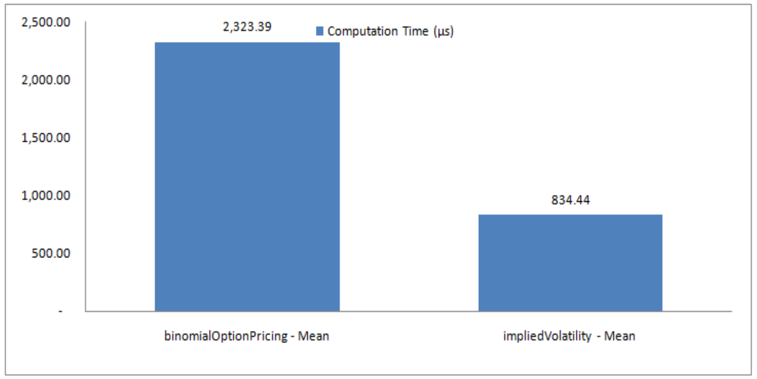


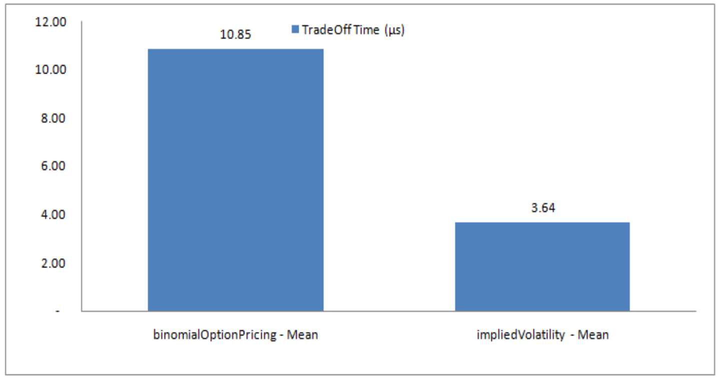
System Behavior Data Set - 10k Bytes

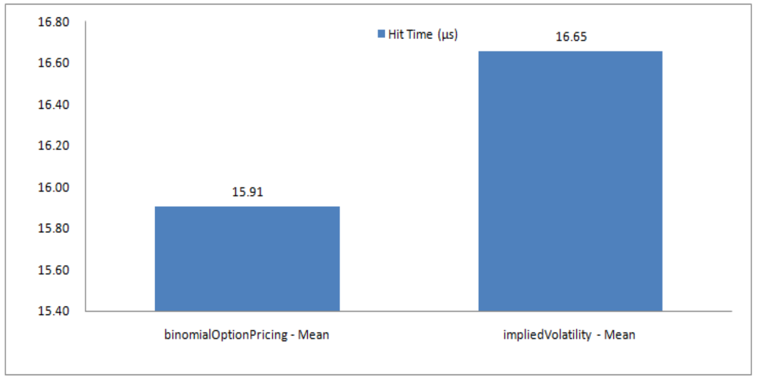
	Miss Time (µs)	Computation Time (µs)	TradeOff Time (µs)	Hit Time (µs)	Number of Miss	Number of Hit	Total Time (µs)	Saved Time (%)	Hits (%)	Alpha break even (%)	Total Energy Saved(kJ)
binomialOptionPricing - Mean	6.47	2,323.39	10.85	15.91	49,078.00	1,232.00	2,283.78	1.70%	2.45	0.74	0.13
binomialOptionPricing - Stdev	12.30	136.59	13.63	11.77	-	-	-	-			
impliedVolatility - Mean	7.60	834.44	3.64	16.65	49,643.00	47.00	844.91	-1.25%	0.09	1.36	(0.03)
impliedVolatility - Stdev	6.55	1,725.36	5.35	4.00	-	-	-	-			

10,000 Calls 10k Bytes





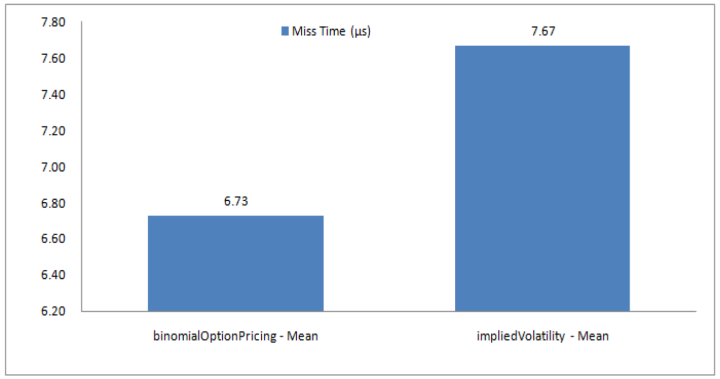


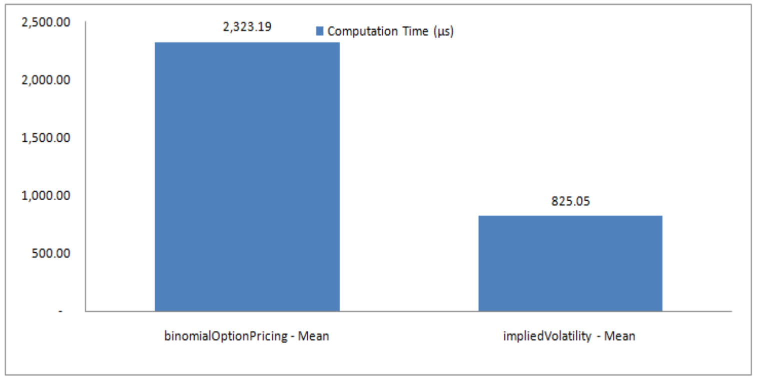


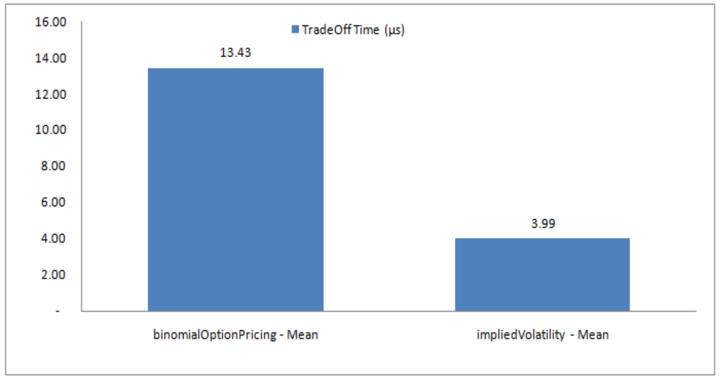
System Behavior Data Set - 15k Bytes

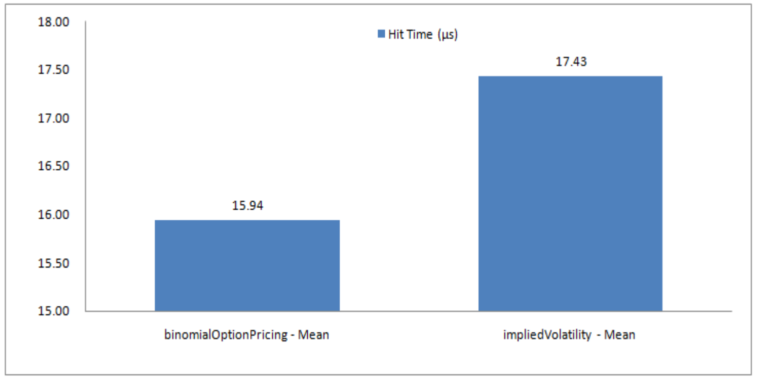
	Miss Time (µs)	Computation Time (µs)	TradeOff Time (µs)	Hit Time (µs)	Number of Miss	Number of Hit	Total Time (µs)	Saved Time (%)	Hits (%)	Alpha break even (%)	Total Energy Saved(kJ)
binomialOptionPricing - Mean	6.73	2,323.19	13.43	15.94	46,329.00	1,691.00	2,261.39	2.66%	3.52	0.87	0.19
binomialOptionPricing - Stdev	6.35	215.95	11.91	8.69	-	-	-	-			
impliedVolatility - Mean	7.67	825.05	3.99	17.43	51,908.00	72.00	835.57	-1.28%	0.14	1.42	(0.03)
impliedVolatility - Stdev	9.04	1,713.54	8.86	4.61	-	-	-	-			

10,000 Calls 15k Bytes





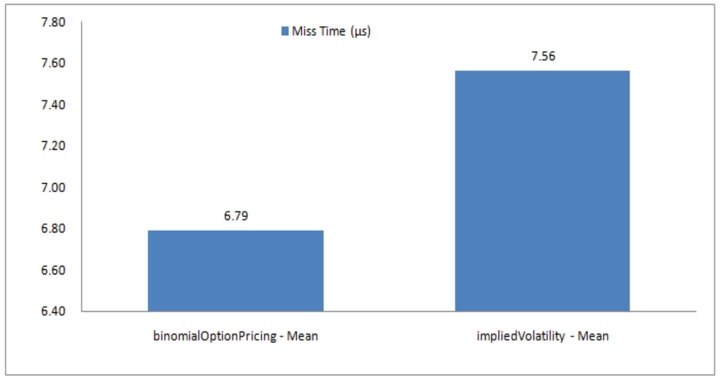


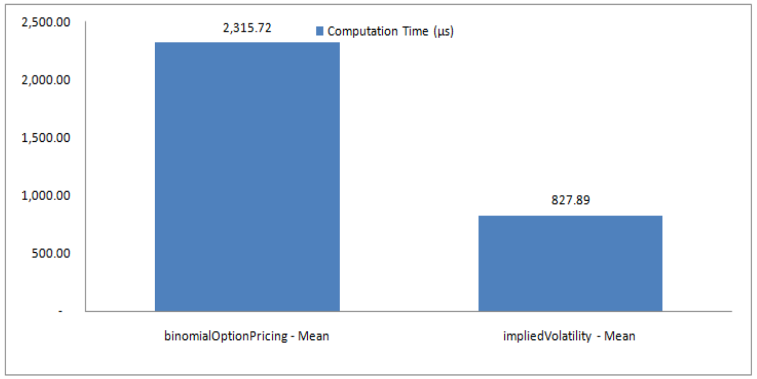


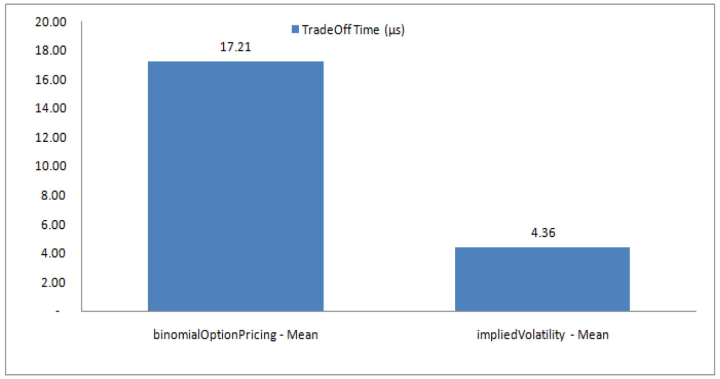
System Behavior Data Set - 20k Bytes

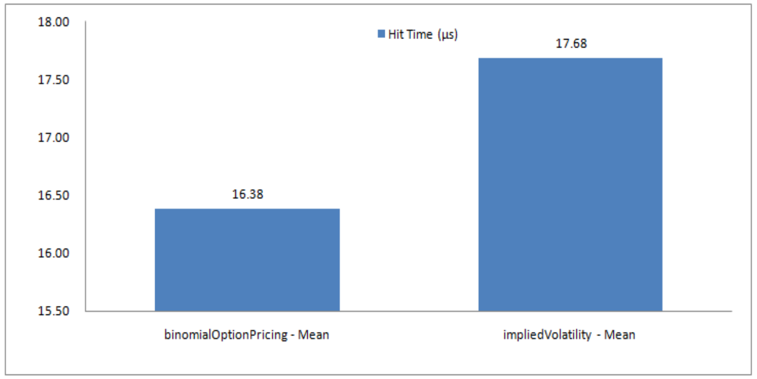
	Miss Time (µs)	Computation Time (µs)	TradeOff Time (µs)	Hit Time (µs)	Number of Miss	Number of Hit	Total Time (µs)	Saved Time (%)	Hits (%)	Alpha break even (%)	Total Energy Saved(kJ)
binomialOptionPricing - Mean	6.79	2,315.72	17.21	16.38	47,624.00	2,406.00	2,227.99	3.79%	4.81	1.03	0.28
binomialOptionPricing - Stdev	6.30	201.03	21.95	14.09	-	-	-	-			
impliedVolatility - Mean	7.56	827.89	4.36	17.68	49,870.00	100.00	838.17	-1.24%	0.20	1.45	(0.03)
impliedVolatility - Stdev	7.02	1,715.33	10.46	9.75	-	-	-	-			

10,000 Calls 20k Bytes



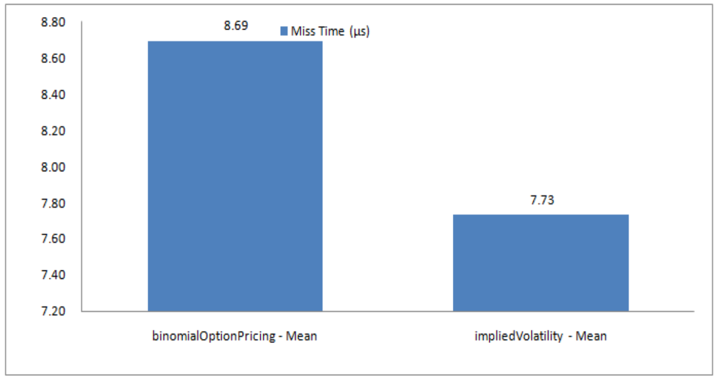


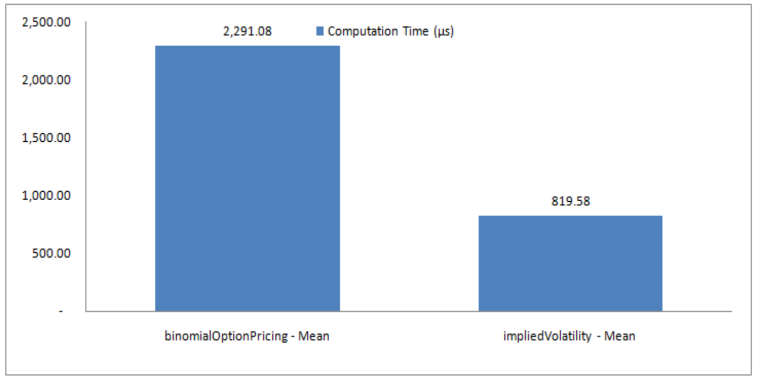


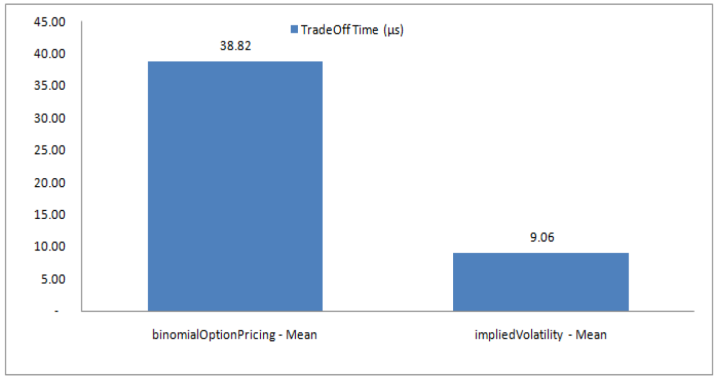


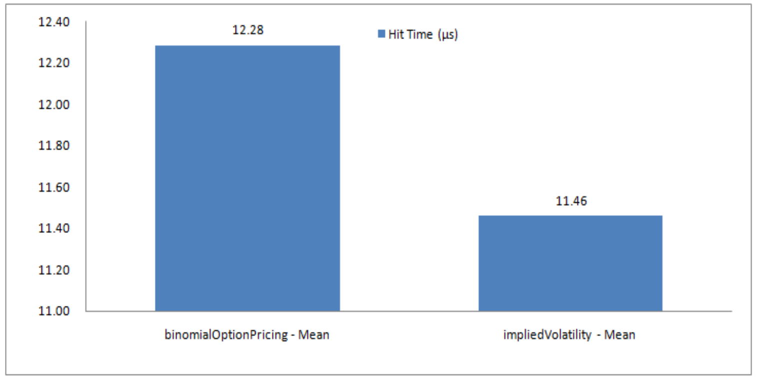
System Behavior Data Set - 50k Bytes

	Miss Time (µs)	Computation Time (µs)	TradeOff Time (µs)	Hit Time (µs)	Number of Miss	Number of Hit	Total Time (µs)	Saved Time (%)	Hits (%)	Alpha break even (%)	Total Energy Saved(kJ)
binomialOptionPricing - Mean	8.69	2,291.08	38.82	12.28	44,699.00	5,441.00	2,086.15	8.94%	10.85	2.04	0.65
binomialOptionPricing - Stdev	6.26	167.14	37.09	6.64	-	-	-	-			
impliedVolatility - Mean	7.73	819.58	9.06	11.46	49,639.00	221.00	832.71	-1.60%	0.44	2.04	(0.04)
impliedVolatility - Stdev	8.48	1,692.93	20.77	2.00	-	-	-	-			
										10,000 Calls	50k Bytes





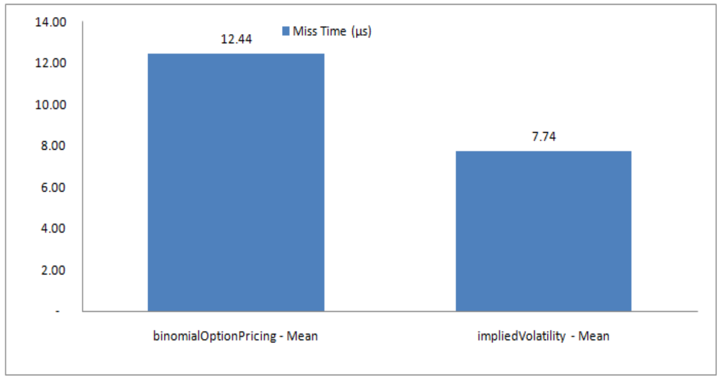


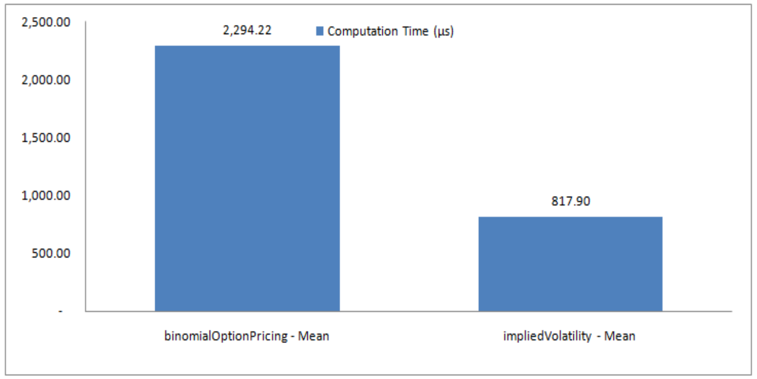


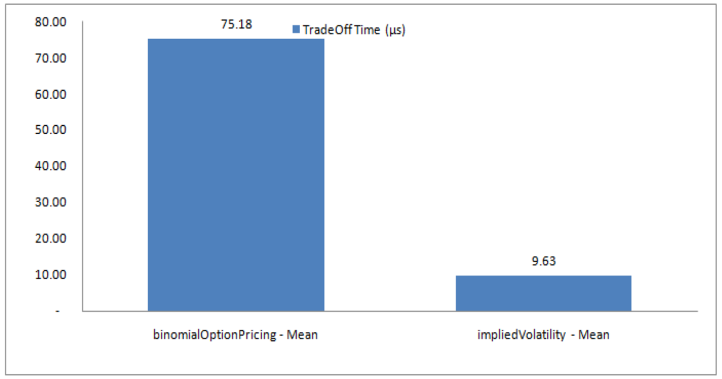
System Behavior Data Set - 100k Bytes

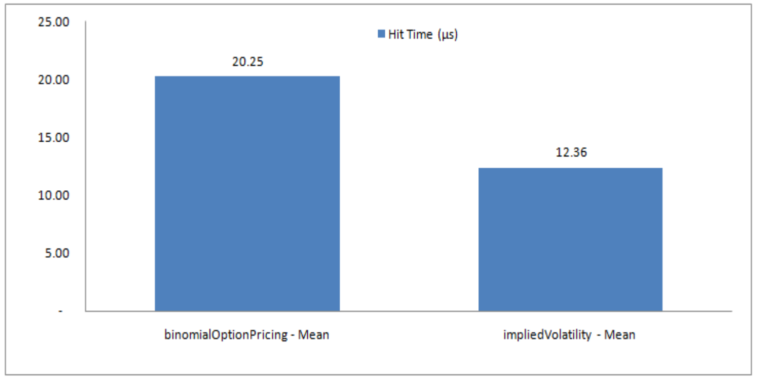
	Miss Time (µs)	Computation Time (µs)	TradeOff Time (µs)	Hit Time (µs)	Number of Miss	Number of Hit	Total Time (µs)	Saved Time (%)	Hits (%)	Alpha break even (%)	Total Energy Saved(kJ)
binomialOptionPricing - Mean	12.44	2,294.22	75.18	20.25	39,817.00	10,273.00	1,897.51	17.29%	20.51	3.71	1.25
binomialOptionPricing - Stdev	10.43	220.72	73.67	11.58	-	-	-	-			
impliedVolatility - Mean	7.74	817.90	9.63	12.36	49,736.00	174.00	832.40	-1.77%	0.35	2.11	(0.05)
impliedVolatility - Stdev	7.46	1,690.04	29.06	9.83	-	-	-	-			

10,000 Calls 100k Bytes





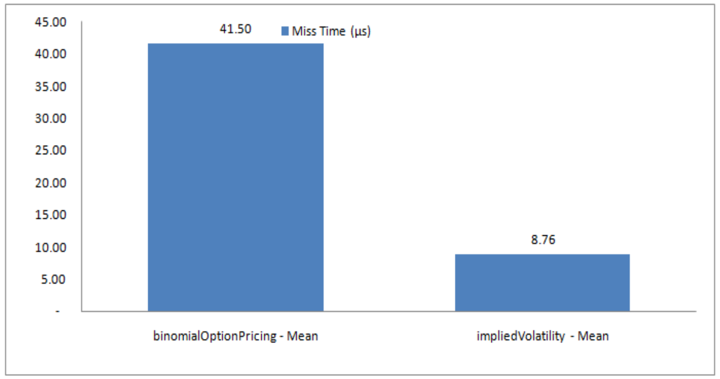


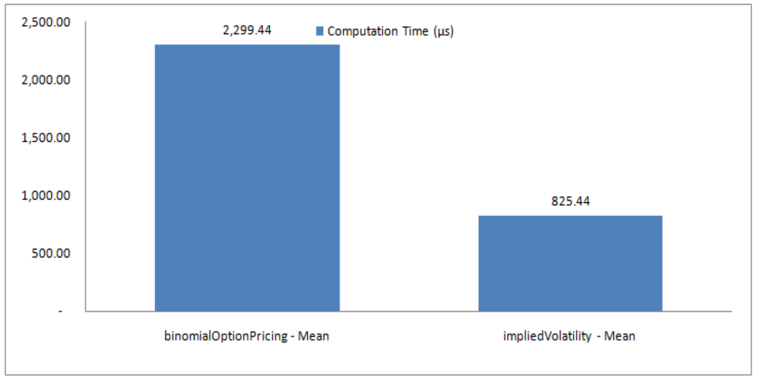


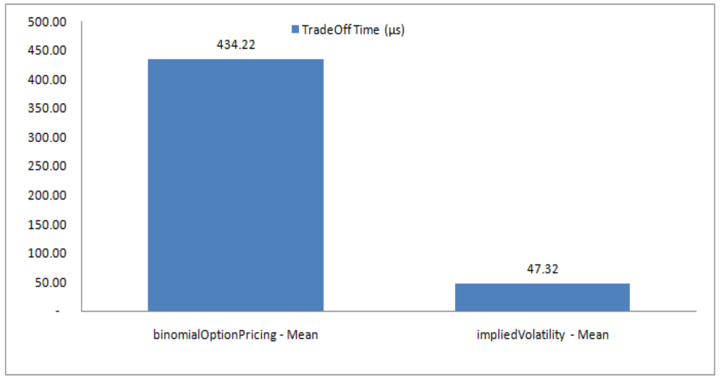
System Behavior Data Set - 500k Bytes

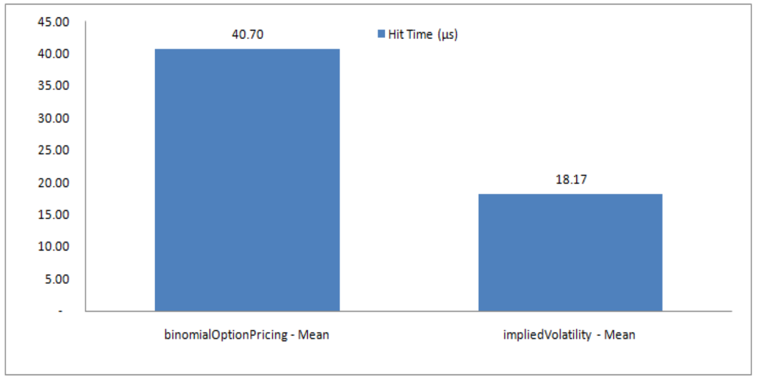
	Miss Time (µs)	Computation Time (µs)	TradeOff Time (µs)	Hit Time (µs)	Number of Miss	Number of Hit	Total Time (µs)	Saved Time (%)	Hits (%)	Alpha break even (%)	Total Energy Saved(kJ)
binomialOptionPricing - Mean	41.50	2,299.44	434.22	40.70	14,006.00	36,344.00	801.35	65.15%	72.18	17.40	4.74
binomialOptionPricing - Stdev	27.54	161.11	488.83	23.00	-	-	-	-			
impliedVolatility - Mean	8.76	825.44	47.32	18.17	48,628.00	1,022.00	863.75	-4.64%	2.06	6.50	(0.12)
impliedVolatility - Stdev	9.93	1,700.17	200.63	11.51	-	-	-	-			

10,000 Calls 500k Bytes



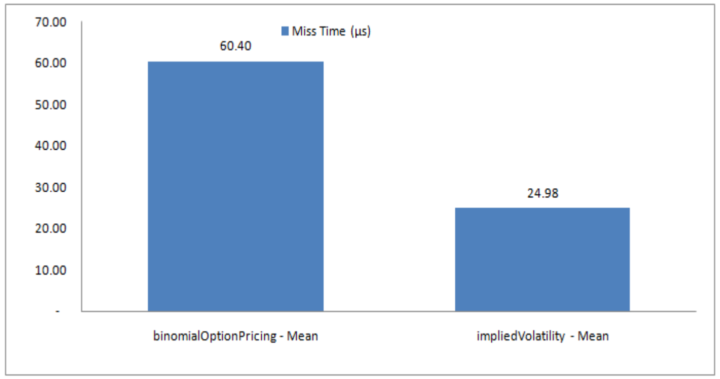


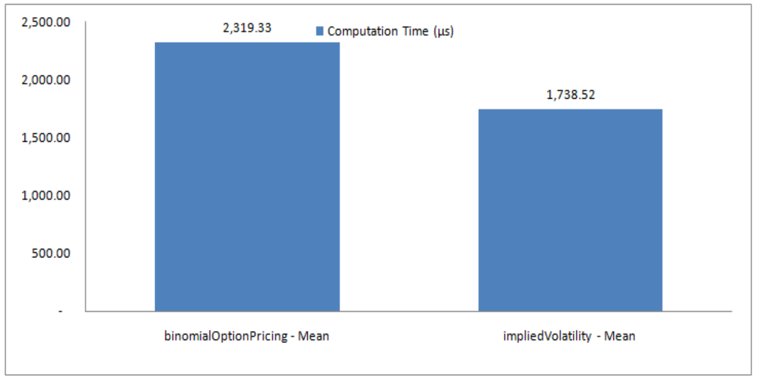


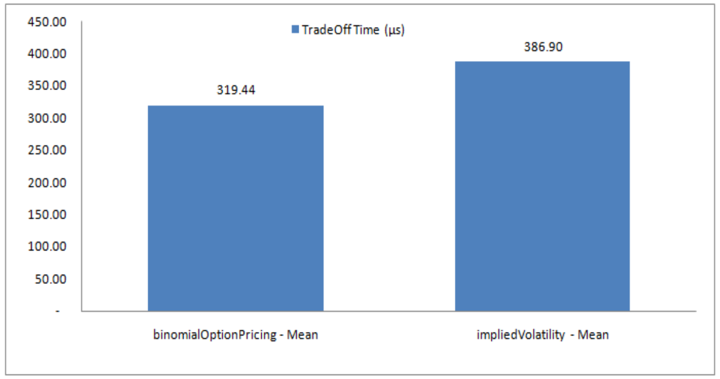


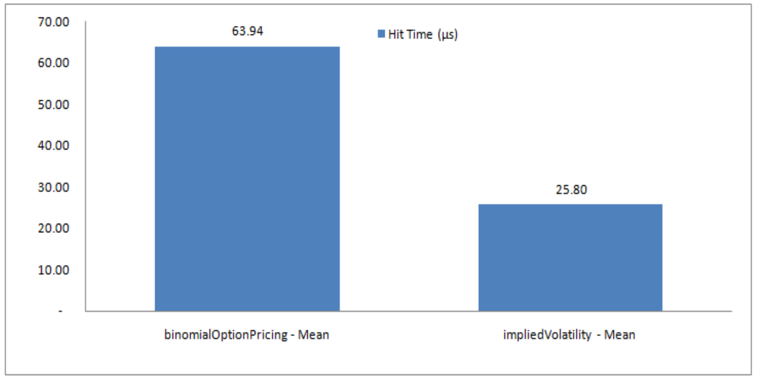
System Behavior Data Set - 1M Bytes

	Miss Time (µs)	Computation Time (µs)	TradeOff Time (µs)	Hit Time (µs)	Number of Miss	Number of Hit	Total Time (µs)	Saved Time (%)	Hits (%)	Alpha break even (%)	Total Energy Saved(kJ)
binomialOptionPricing - Mean	60.40	2,319.33	319.44	63.94	5,825.00	43,645.00	374.24	83.86%	88.23	14.41	6.05
binomialOptionPricing - Stdev	34.54	611.64	574.94	33.32	-	-	-	-			
impliedVolatility - Mean	24.98	1,738.52	386.90	25.80	14,782.00	35,748.00	647.33	62.77%	70.75	19.39	3.46
impliedVolatility - Stdev	14.51	2,483.98	639.30	16.87	-	-	-	-			
										10,000 Calls	1M Bytes



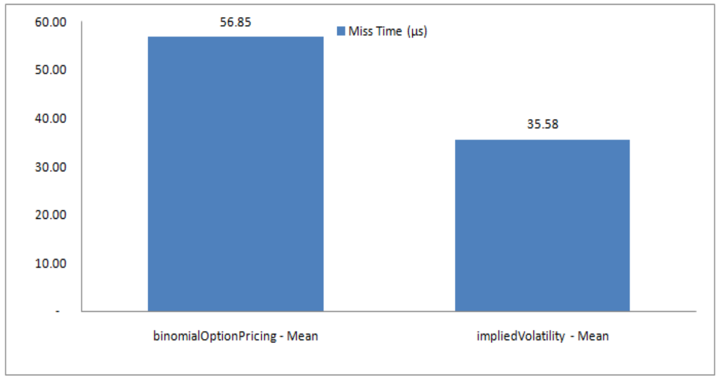


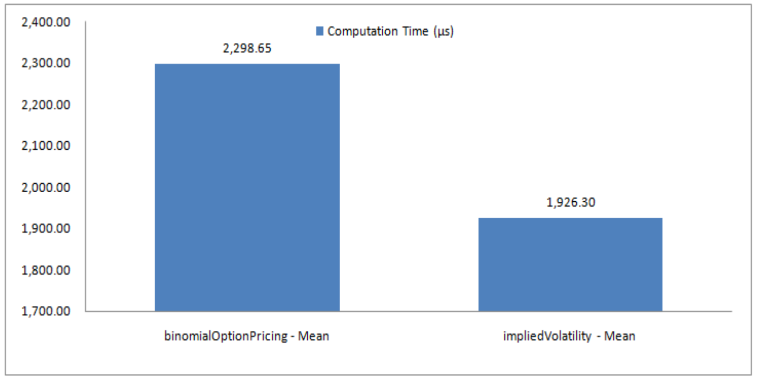


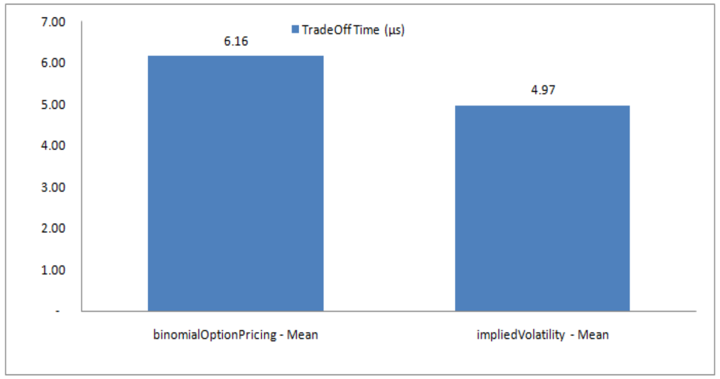


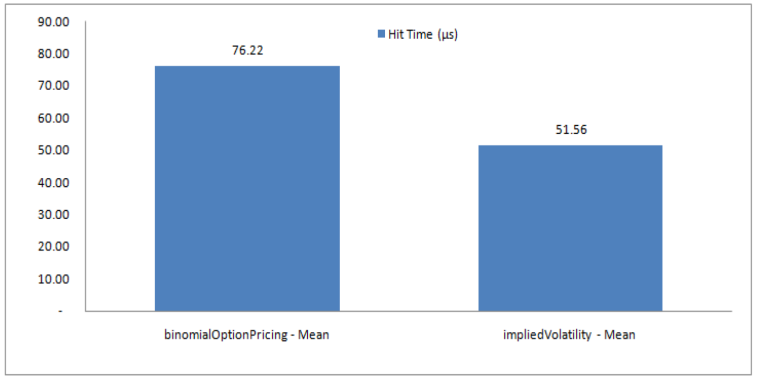
System Behavior Data Set - 100M Bytes

	Miss Time (µs)	Computation Time (µs)	TradeOff Time (µs)	Hit Time (µs)	Number of Miss	Number of Hit	Total Time (µs)	Saved Time (%)	Hits (%)	Alpha break even (%)	Total Energy Saved(kJ)
binomialOptionPricing - Mean	56.85	2,298.65	6.16	76.22	3,121.00	47,019.00	218.48	90.50%	93.78	2.76	6.55
binomialOptionPricing - Stdev	49.51	189.14	36.32	32.08	-	-	-	-			
impliedVolatility - Mean	35.58	1,926.30	4.97	51.56	1,805.00	48,055.00	120.90	93.72%	96.38	2.12	5.66
impliedVolatility - Stdev	24.88	2,951.42	6.37	31.55	-	-	-	-			
										10,000 Calls	100M Bytes









Bibliography

[1] E. Williams. Energy intensity of computer manufacturing: Hybrid assessment combining process and economic input-output methods. *Environ. Sci. Technol.*, (38):6166–6174, 2004.

[2] Berkeley National Lab. Lawrence. Optimization of product life cycles to reduce greenhouse gases in california. *Report for California Energy Commission.*, (CEC-500-2005-110-F), 2005.

[3] Industrial Research and Development Corporation. Personal computers (desktops and laptops) and computer monitors. *Report for the European Commission, August 2007.*, 2007.

[4] S. Murugesan. Harnessing green it principles and practices. *IT Professional.*, 10:24–33, 2008.

[5] Chiara Francalanci. Green it: Sfide e opportunita. *Mondo Digitale.*, pages 36–42, 2008.

[6] N. L. B. Levitin Margolous. *Physica D.*, page 188, 1988.

[7] Poletto Erick B. Fiorelli, Ricardo A. Data center energy efficiency: Analysis and test of energy consumption benchmark tools. page 44, 2009.