POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAM IN INFORMATION TECHNOLOGY

A METHODOLOGY TO IMPROVE THE ENERGY EFFICIENCY OF SOFTWARE

Doctoral Dissertation of:
Marco Bessi

Advisor:
Prof. Chiara Francalanci

Co-advisor:
Prof. Giovanni Agosta

Tutor:
Prof. Barbara Pernici

The Chair of the Doctoral Program:
Prof. Carlo Fiorini

2013 – XXVI

POLITECNICO DI MILANO
Dipartimento di Elettronica, Informazione e Bioingegneria
Piazza Leonardo da Vinci, 32 20133 — Milano (MI)

To my parents.

To my wife.

If you always make the right decision,
the safe decision,
the one most people make,
you will be the same as everyone else.
[Paul Arden]

ACKNOWLEDGEMENTS

The printed pages of this dissertation hold far more than the culmination of years of study. These pages also reflect the relationships with many generous and inspiring people I have met since beginning my doctoral work.

This thesis would not have been possible without the support of my principal supervisor, Prof. Chiara Francalanci. The good advice of my second supervisor, Prof. Giovanni Agosta, has been invaluable on academic level, for which I am extremely grateful. Without his supervision and constant help this dissertation would not have been possible.

I would also like to thank my loving parents, Anna e Paolo. They were always supporting me and encouraging me with their best wishes.

Obviously, I would like to thank all the SboccoClub friends for helping me get through the difficult times, and for all the emotional support, camaraderie, and entertainment they provided. I am indebted to my many student colleagues for providing a stimulating and fun environment in which to learn and grow. I am especially grateful to Leonardo, who as a good friend, was always willing to help and give his best suggestions.

Last but not least, I would like to thank my wife, Veronica. She was always there cheering me up and stood by me through the good and bad times. I love you.

Marco

Milano, January 2014

ABSTRACT

WHEREAS hardware is physically responsible for power consumption, hardware operations are guided by software, which is indirectly responsible of energy consumption. Research on Information Technology (IT) energy efficiency has mainly focused on hardware. This thesis focus on software energy consumption with the goal of providing a methodology to identify and reduce energy inefficiencies.

In our approach, we estimate the energy consumption of application software independent of the infrastructure on which the software is deployed. Our methodology make a distinction between the usage of computational resources and the unit energy consumption of each resource. In this way, usage of resources and unit energy consumption can be measured independently. As thoroughly discussed in the thesis, this separation allows the definition of energy consumption benchmarks for homogenous clusters of transactions or software applications. These metrics can be used to define classes of energy efficiency within each cluster, thus enabling the comparison of the energy efficiency of similar applications/transactions.

Benchmarks are used to identify energy inefficient software applications. The thesis proposes a memoization-based approach to improve the energy efficiency of inefficient software that does not require a refactoring of code. Optimizing code has a direct beneficial impact on energy efficiency, but it requires domain knowledge and an accurate analysis of the algorithms, which may not be feasible and is always too costly to perform for large code bases. We present an approach based on dynamic memoization to increase software energy efficiency without a need for a direct optimization of existing code. We analyze code automatically to identify

a subset of pure functions that can be tabulated and automatically store results. The basic idea of our memoization-based approach is to fetch previously computed results from memory to avoid computation. This idea raises a number of research challenges. The thesis discusses a software framework that provides a tool-set to apply memoization to any software application tested on a sample of financial functions provided by a large Italian corporation that have accepted to participate in the empirical testing of our methodology as a pilot case set. Empirical results show how our approach can provide significant energy and time savings at limited costs, with a considerably positive economics impact.

SOMMARIO

ORMAI da qualche anno si parla diffusamente dell'efficienza energetica dell'IT, ma raramente si affronta il problema del ruolo rivestito dal software nel determinare il consumo energetico dell'IT. Considerando che l'hardware è fisicamente responsabile per il consumo di potenza e che le operazioni hardware sono guidate dal software, allora possiamo affermare che il software è indirettamente responsabile del consumo di energia. La ricerca relativa all'efficienza energetica nel mondo IT si è concentrata principalmente su hardware. Questo lavoro di tesi pone l'attenzione sul consumo di energia del software con l'obiettivo di fornire una metodologia per identificare e ridurre le sue inefficienze energetiche.

E' stata definita quindi una metodologia per la stima del consumo di energia del software applicativo indipendentemente dall'infrastruttura su cui viene installato il software. Il nostro approccio si basa sulla distinzione tra l'utilizzo di risorse computazionali imputabili all'uso dell'applicazione e il consumo unitario di ogni risorsa utilizzata. In questo modo, l'utilizzo delle risorse e il consumo unitario energetico possono essere misurati in modo indipendente. Questa separazione permette la definizione di parametri di riferimento del consumo di energia per cluster omogenei di transazioni o di applicazioni software. Questi parametri possono essere utilizzati per definire classi di efficienza energetica all'interno di ciascun cluster, permettendo così il confronto dell'efficienza energetica di transazioni o operazioni similari.

I parametri di riferimento precedentemente calcolati sono quindi utilizzati per identificare le applicazioni software inefficienti dal punto di vista energetico. Questo lavoro di tesi propone un approccio basato sulla me-

moizzazione dinamica per migliorare l'efficienza energetica dei software inefficienti, senza richiedere un refactoring di codice. L'eventuale ottimizzazione manuale del codice di un'applicazione ha un impatto positivo diretto sulla efficienza energetica, ma richiede la conoscenza di dominio e di una accurata analisi degli algoritmi, che non sempre è fattibile ed è molto costosa da eseguire per le grandi basi di codice. Il nostro approccio basato sulla memoizzazione permette di migliorare l'efficienza energetica del software senza la necessità di una ottimizzazione diretta del codice esistente. Il codice viene analizzato automaticamente per identificare un sottoinsieme di funzioni pure che possano essere tabulate per memorizzarne automaticamente i risultati. L'idea di base del nostro approccio è quello di recuperare dalla memoria i risultati precedentemente calcolati per evitare l'uso della CPU per effettuare calcoli aggiuntivi. La tesi presenta inoltre il framework che implementa la metodologia fornendo un set di strumenti per applicare la memoizzazione a qualsiasi applicazione software. Il framework è stato testato su un campione di funzioni finanziarie fornite da una grande azienda italiana che ha accettato di partecipare come un caso pilota alla verifica empirica della nostra metodologia. I risultati mostrano come il nostro approccio può fornire un notevole risparmio energetico e di tempo a costi contenuti, con un impatto economico molto positivo.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

In the last ten years IT systems have grown very fast, with a consequent growth of the IT energy consumptions. This growth has raised a number of considerable issues. First of all, energy costs have dramatically increased and their impact on the overall IT infrastructural costs has become even more significant. Second, energy requirements represent one of the scalability issues of datacenters, since providers have difficulties at supplying data centers with all the required energy. Moreover, information technology contributes strongly to the green-house CO_2 emissions. All these issues belong to a new filed of study: Green IT.

Although software does not directly consume energy, it affects the energy consumption of IT equipments. Software applications indicate how information should be elaborated and to some extent guide the use of hardware. Consequently, software is indirectly responsible of energy consumption. Despite this, software engineering focuses on software performances and quality. Energy efficiency is not even included among software quality metrics [48]. Most enterprises focus on the trade-off between cost and quality, while neglecting software energy efficiency. Therefore there is a need to analyze, quantify and optimize application software from the point-of-view of the energy efficiency.

The literature explains how optimizing software algorithms has a direct and potentially beneficial impact on energy efficiency. However, there is broad evidence of the practical hurdles involved in optimizing software algorithms, tied to the technical skills of programmers, to the role of domain knowledge, and to the need for massive code refactoring and related costs. This thesis focus on software energy consumption with the goal of providing a methodology to identify and reduce energy inefficiencies at limited costs.

The energy consumption of IT has attracted the attention of both academic and industrial communities in the last years. Green IT raises research challenges at different infrastructure levels, including the selection of hardware devices, data centers design, management and usage practices, and software design and development. In our previous researches [15] [16] [17] is shown the importance of software energy efficiency, as the ultimate cause for the energy consumption of all other infrastructural levels. IT managers seem to believe that software per se has a very limited impact on the energy consumption of real-word systems. Their claim is that although the applications have different designs, they do not differ significantly from one another in terms of hardware requirements and, thus, energy efficiency. It is generally accepted that the operating system can make the difference in energy consumption, not the application software itself [49]. However, as a matter of fact these beliefs have never been empirically verified using a systematic scientific approach and there is no clear evidence as to whether application software has a tangible impact on energy consumption.

We propose a methodology to estimate the energy consumption of application software independent of the infrastructure on which the software is deployed. Our methodology makes a distinction between the usage of computational resources by an application and the unit energy consumption of computational resources. In this way, usage of computational resources and unit energy consumption can be measured independently. The usage of computational resources includes the usage of processor and input/output (I/O) channels to execute transactions. Both measures can be easily obtained with widely used testing platforms. Unit energy consumption can be measured just once for each class of devices or extracted from their datasheets. Our methodology can support the definition of benchmarks of energy consumption for classes of functionally similar transactions or applications. These values can be used to define classes of efficiency within a given sample of applications or to compare the energy performance of similar applications.

These benchmarks are used to identify energy inefficient software ap-

plications. In this thesis we present a methodology that applies memoization techniques to improve the energy efficiency of inefficient application software. Memoization is a programming technique that caches the results of a software program in memory. Memoization increases energy efficiency when storing and fetching a value consumes less energy than executing the corresponding software program. In general, processing a complex computation-intensive function requires more energy than caching results and reading them when needed, especially if the function is frequently called with the same input values. Furthermore, these benefits can be reaped without a need for code inspection and redesign, as they are the result of an intelligent management of memory resources at run time. Memoization has been extensively studied by the literature and has been applied in different contexts, including dynamic programming [19] [33] and incremental computation [25] [28]. However, applying memoization to improve the energy efficiency of application software raises new research challenges. First, a restrictive definition of the concept of pure function is required to avoid manual inspection of code and enable large scale applicability. Pure functions are good candidates for memoization. Second, the benefits of memoization depend on the precision of results: the greater the precision, the larger the size of memory required to store all possible results of a function. While this issue is neglected in existing literature, in the context of application software precision represents a typical item of a Service Level Agreement (SLA) and can be tuned at run time. Our methodology obtains an energy-efficient management of memory resources in three steps: 1) pure functions are identified as good candidates for memoization, 2) a subset of initial candidate functions is selected to make sure that each selected function can improve energy efficiency when memoized, 3) available memory is allocated to different functions in order to minimize overall energy consumption based on a set of SLAs. Note that the second step includes an empirical performance model supporting the runtime estimate of the energy benefits from the memoization of each candidate function. The model includes two important parameters: a) the statistical distribution of the input values of each candidate function, b) the impact precision requirements on the amount of memory required for the memoization of the function. Both parameters are strongly affected by the application context and represent important determinants of the overall feasibility of the methodology.

We have developed a software suite to test our approach. The methodology is tested on a set of well-known financial functions. The experimental campaign is executed with automatically generated input workloads based

on real parameters, including the range, mean, and standard deviation of input parameters. Both the code and the workload parameters have been provided by a large Italian corporation that have accepted to participate in the empirical testing of our methodology as a pilot case set. Empirical results show how our approach can provide significant energy and time savings at limited costs, with a considerably positive economics impact.

The presentation is organized as follows. Chapter 2 surveys the most important works related to the recent researches on energy efficiency in the IT field, also considering memoization approaches and software applications using these techniques. Chapter 3 describes an energy benchmarking approach to estimate the energy consumption of application software independent of the infrastructure on which the software is deployed. This separation allows the definition of energy consumption benchmarks for homogenous clusters of transactions or software applications. Then, benchmarks are used to identify energy inefficient software applications. Chapter 4 proposes a memoization-based approach to improve the energy efficiency of inefficient software that does not require a refactoring of code. Our approach increase software energy efficiency without a need for a direct optimization of existing code. We also propose a mathematical model to estimate the effectiveness of the memoization approach and the policy for the memory allocation in order to maximize the effectiveness of the energy efficiency of the approach. Chapter 5 describes the proposed solution and its implementation. The Green Memoization Suite (GreMe) allows the analysis and the code modification of the bytecode. Then, the modified bytecode can be executed implementing the memoization technique. Chapter 6 presents and discusses the experimental results obtained from the application of the approaches described in Chapter 5. Finally, Chapter 7 draws some conclusions and proposes future research directions.

CHAPTER 2

STATE OF THE ART

The growth experienced by the Information Technology (IT) industry, especially in the last decade, has resulted as an impressive demand in terms of energy needs, and hence environmental impact. However, the energy efficiency of such technologies is far below expectations, resulting in non negligible energy wastes. For these reasons, additional studies and techniques have to be developed to improve the electric consumption of these devices.

This Chapter discusses the recent researches on energy efficiency in the IT field, also considering memoization approaches and software applications using these techniques. In Section 2.1, we introduce the Green IT and Green Computing concepts and the fundamental aspects are described. Section 2.3 illustrates the impact of the software on Green IT. Section 2.4 describes the recent memoization-based approaches and techniques.

2.1 Green IT: overview

Nowadays, Green IT is one of the most relevant and discussed topics in the Information Technology industry. Two different kinds of costs are brought by: the energetic and the maintenance ones. The increasing need for more

electric power, related not only to servers and infrastructures but for example also to conditioning systems, has a direct impact on the environment; in fact, more power translates in more carbon emissions (due the industrial processes that provide electricity) and hence influences in a direct way the global warming.

2.1.1 Energy consumption overview

[40] provide an insight regarding the percentage of energy that the main components of a data center require, identifying the main contributions from servers (56%) and cooling systems (30%), while the rest of the power is accountable to all the remaining components (network, lights, etc.). Perhaps the most known and impressive research regarding the impact of the IT, in terms of carbon emissions, is presented by Gartner Inc [4]. According to it, up to 2% of the total emissions of carbon dioxide is related to the ICT. About the 23% of this quantity, according to Gartner, is accountable to data centers, and hence is directly related to the power required by servers and cooling systems.

The energy required to power all the world's computers, data storage, and communications networks is expected to double by 2020, according to a McKinsey & Company analysis [11]. This would increase the total impact of IT technology, in terms of global carbon emissions, up to 3%.

2.1.2 Energy efficiency overview

An important factor to consider when dealing with power consumption is the energy efficiency. This measure defines the ratio between the use of a device (for example, a common server) and the power required to achieve it. Of course, greater ratios correspond to better utilization, and hence to a better utilization of electrical consumption; however, in modern technologies, there is not a proportional relationship between workload of a machine and its power required. In [9], the authors provide an overview about the typical utilization of a server, and the energy associated to it. Figure 2.1 shows a representation of the average processor utilization, resulting from the observation of more than 5000 servers for a period of six months.

Although the service provided could modify the shape of the distribution, it is evident how the servers are rarely idle or working at full capacity, operating most of the time between 10-50% of their maximum level. This is expectable, since services that average close to 100% of the servers capacities could result in difficulties to meet throughput and latency agreed in the Service Level Agreement (SLA), for example due to software-hardware

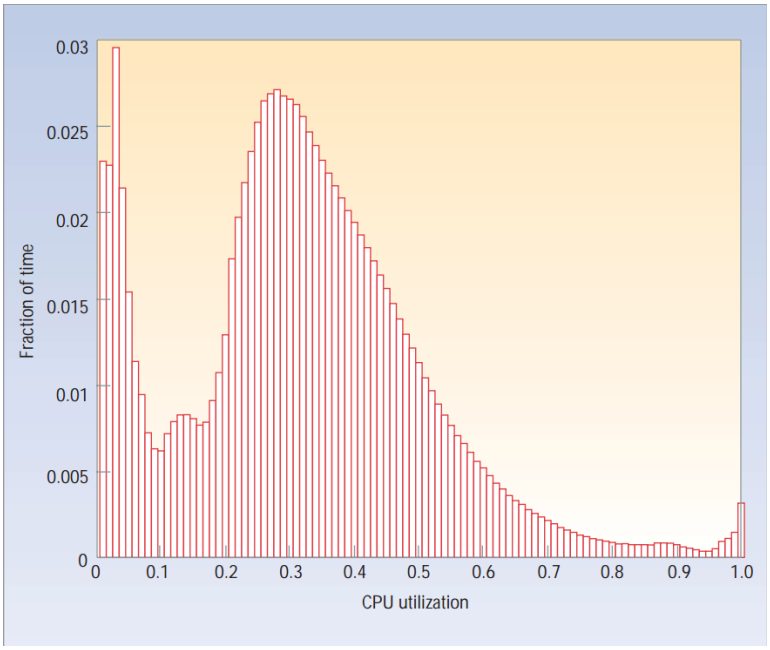


Figure 2.1: *Average CPU utilization of more than 5,000 servers during a six-month period. Servers are rarely completely idle and seldom operate near their maximum utilization, instead operating most of the time at between 10 and 50 percent of their maximum utilization levels.*

failures or need for maintenance. On the other hand, a higher time in idle or at very low capacity could represent a substantial waste of energy, since the data center is evidently over dimensioned for the service provided. Having identified the common operating region, it is possible to observe the energy efficiency at these levels. Figure 2.2 compares the server power usage to the correspondent energy efficiency; the values of this measure are astonishing low, establishing below than 40% in the 20-30% utilization range.

This translates in an energy waste greater than 60% for the major part of the time. Moreover, even when the server is in idle, it still consumes about half of the energy required when working with full load. The energy efficiency tends to assume better values when approaching the maximum capacity of the machine, but as already said such intervals are almost never met for a well-designed service. Moreover, since in a common server only 20% of the energy is required from the computation, while 80% accounts for other elements (alimentation, fans, memory, etc.), for each Watt actually used for computation the processor consumes 5W, while the entire server absorbs 16W and the data center a total around 27W [42].

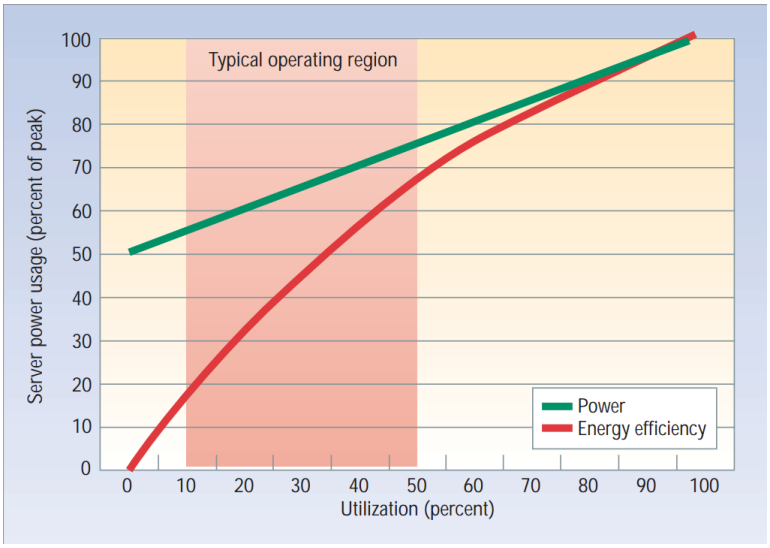


Figure 2.2: *Server power usage and energy efficiency at varying utilization levels, from idle to peak performance. Even an energy-efficient server still consumes about half its full power when doing virtually no work.*

The theme of energy efficiency is also explored in [13], which deeply analyzes the contribution of different components at server level, as well as data center level. In particular, it underlines the need for new practices to improve such efficiency, or face a constant decrease of such ratio in the near future (meaning even worse management of electrical power). Moreover, it shows how from 2008 the cost for a server has been overcome by the one relative to the energy required, not only in terms of the machine itself, but also regarding external factors (such as the need for cooling). The relatively cheap option of buying more servers to obtain higher throughput is hence contrasted by the increasing cost of energy, resulting in reduced profits and scalability problems for the companies that invested in such model.

[38] provide an overview of the energy efficiency of different components of a data center. While only a few elements have values of such ratio over (or even approaching) 50%, it is astonishing how the common processors fails to reach even 1% of energy efficiency; for a complete list of the component considered in the paper, refer to Table 2.1. Techniques such as Dynamic Voltage and Frequency Scaling (DVFS) [31] could improve the result, but still it remains far beyond an acceptable level.

These figures provide an overview on how the typical model of IT expansion is unsustainable, both in terms of cost and energy consumption, as well as towards environment pollution. Solutions at various levels must

Table 2.1: *Estimated energy efficiency of common IT components, with respect to the maximum, theoretical efficiency*

Component	Estimated energy efficiency
Infrastructure	50%
System	40%
Server	60%
Processor	0.001%
Software	20%
Network	10%
Database	60%
IT use practices	30%

be adopted to reduce the energy needed from modern systems, and hence accomplish the double benefit of increase energy efficiency and decrease the pollution generated by energy needs. So, the focus should be not only towards the search for more, innovative and low-impact forms of energy production, but especially related to the better utilization of existing technologies, to improve the energy efficiency and reduce the global demand of power required.

2.2 Green computing

The term Green Computing refers in general to environmentally sustainable computing. A traditional definition is given by San Murugesan in [38]. In the paper, he defines Green Computing as *"the study and practice of designing, manufacturing, using, and disposing of computers, servers, and associated subsystems - such as monitors, printers, storage devices, and networking and communications systems - efficiently and effectively with minimal or no impact on the environment"*. From this definition it is clear that the areas of applications of such discipline include all the life-cycle of the IT products, and more specifically production, use and disposing.

First of all, the identified devices (take for example a common PC) must be designed and manufactured in such a way that the environmental impact is as modest as possible; this includes not only the ability to manage factories and production chains in the most convenient way feasible, but also to prolong the life of such devices as much as possible. In fact, the extension of the equipment lifetime is one of the biggest contribution to Green Computing. The advantages of having a PC that is easily upgradable to a better one, instead of realizing a completely new machine, are very evident in terms of environmental impact. Moreover, Gartner at [37] stated

to *"Look for product longevity, including upgradability and modularity"*, making clear once again the importance of the cited aspects. Moreover, minimizing the use of non-biodegradable components and encouraging the use of sustainable resources are additional factors that could really improve the environmental impact.

The disposing of electric and electronic equipment is another field that, if not addressed in an adequate manner, can deeply impact the environment in terms of pollution. In fact, lots of toxic materials (for instance lead, mercury, chromium) need specific way of disposing due their highly dangerous nature for the environment. For these means, the European Union has developed in 2002 a specific law, under the name of Waste Electrical and Electronic Equipment Directive (WEEE Directive), that regulates the process of disposing of such material, from collections to effective dispose. Moreover, it regulates another important factor that is very relevant in this field: the recycling. Although the re-utilization of some components, such as hard drive and memory disks, could still arise privacy issues (due the fact that some data could still be present on such devices after the dismission from the original owner), many other parts (including batteries, ink cartridges and generic computing supplies) can be recycled through certain retail outlets, such as [5].

The last area to consider is the utilization of the equipment constituting the IT industry. More in general, two main aspects that deeply influence the environmental sustainability can be considered: the software optimization (known also as Green Software, and better described in the following section) and the power management. The latter is a key factor in terms of energy savings (and hence reduction of carbon dioxide emissions related to energy production), and must be carefully addressed to avoid waste of energy that could become critical, especially for medium-large data centers (which constitute the main element in terms of power demand).

2.3 Green software

An often neglected factor, even when dealing with required power issues, is the one referred to the power consumption attributable to generic application software. Even if software does not consume energy in the traditional sense of the meaning, it is however true that its execution affects the global power absorbed by the entire system that hosts it. Moreover, a sort of propagation can be verified, in terms of power overhead required, due to power-inefficient software. For example, an application could run temporally inefficient algorithms that increase both the time required from the task to be

completed, as well as the power needed by the processor. More power to the cpu means more heat generated by the component, and hence translates in more cooling-power required to ensure the stability of the temperature in which the system operates, thus resulting in more power needed also by the cooling system. This simple example shows how the execution of generic software can deeply impact not only the machine on which it runs, but all the environment in which the system resides, and hence such issue must be carefully addressed by programmers and administrators. Thus, the first needs is to determine the energy consumption of applications running on servers and to benchmark them in order to understand where optimization are needed.

2.3.1 Software benchmarking

Even though hardware infrastructure is physically responsible for energy consumption in data centers, the role of software cannot be neglected. Software is the first cause for energy consumption, as it drives the operations performed by the processor and thus influences the consumption of all the layers of a computer infrastructure. In previous work, we found that the Management Information Systems (MIS) application layer impacts on energy consumption up to 70 (w.r.t. idle consumption) and that different MIS applications satisfying the same functional requirements may consume significantly different amounts of energy (differences up to 145%) [15].

There are several consolidated methodologies to measure hardware energy consumption and efficiency. Hardware performs elementary operations, so it is easy to measure unit energy consumption through metrics such as Joule per FLOP (Floating Point Operation) or MIPS (Million Instructions Per Second) / Watt. Energy efficiency is usually assessed by stressing servers and PCs with benchmark workloads that gradually increase the usage of the processors and by measuring power with specific hardware kits. SPEC [22] and TPC [20] are widely used benchmarks provided by independent organizations. In contrast, the estimation of software energy efficiency remains largely unexplored. Software engineering literature proposes several metrics for software quality. However, these metrics never include power consumption [17]. Some research works [39] have analyzed the relationship between quality metrics and power consumption, in the context of embedded systems. Not surprisingly, the 50 ISO software quality parameters [48] do not include energy efficiency. A significant body of literature has focused on time performance [30] [32] [44] [45] [50]. To the best of our knowledge, no previous studies relate time performance to

the energy efficiency of application software. In previous literature methodologies can be found to estimate software energy efficiency [18]. For example, [27] investigates low power embedded systems and introduces accurate power metrics to drive the hardware/software co-design. These works are limited to embedded systems and cannot be extended to business applications, such as ERPs. In [18], software energy efficiency is addressed and innovative energy consumption metrics are proposed. Their metrics can be extracted from the flow graph of the software program. The main limitation of this work is that metrics have been validated on matrix algebra and multimedia programs, whose execution flows are easily predictable. In [15], we have proposed a methodology to measure MIS software energy consumption. This methodology consists in defining software benchmark workloads and measuring the total energy consumed by the system to execute a given workload. The main drawback of this approach is that experimental campaigns have to be executed on the same hardware infrastructure. For this reason, applications can be compared only if they are executed on the same machine. In (omitted-reference), we have extended our methodology to measure the energy efficiency of MIS applications. This is obtained by defining classes of functionally similar applications (e.g., ERPs, spread sheets, etc.) and measuring the energy consumption associated with the execution of the same benchmark workload. The comparison of normalized values within the sample of applications allows the assessment of energy efficiency. This methodology provides an assessment that depends on the specific hardware infrastructure used for testing. More generally, there are two main open issues in the literature. First, software causes energy consumption, but the hardware infrastructure is responsible for the physical absorption of power. Thus, software energy efficiency metrics provide different results on different hardware platforms. Second, while the elementary tasks executed by hardware devices are easy to identify (e.g., operations, instructions, I/O exchanged), the same is not true for application software. In particular, the output of software depends on the input and cannot be easily standardized. To the best of our knowledge, no hardware independent benchmarks methodology has been previously proposed for software energy efficiency.

2.3.2 Green optimization techniques

To increase the energy efficiency of software applications, some green optimization techniques can be applied. Traditional research focuses on software for embedded systems, in which power consumption is critical. Max-

imizing the energy source and hence the life of these components is a key factor to develop reliable and efficient embedded and/or distributed systems. Researches on Low Power Software for embedded systems focus on the software design for specific, integrated systems, ensuring energy savings and hence more competitive solutions. Additional techniques can also be adopted to improve energy efficiency for this kind of systems; among them there are Context Awareness (the ability to respond to changes in the environment on which they operate, to ensure additional energy savings), Code Compressions (reduce the memory size required by the software, which determine less accesses to the memory and hence less energy required) and general optimizations that are usually focused on the memory (another critical factor for embedded systems). From the point of view of common application software, instead, not much has been proposed, due the fact that usually system administrators fail to identify it as a source of potential energetic inefficiency. The fact that often there are not immediate and tangible benefits in investing in such areas discourage the adoption of techniques of code design and optimization that can, instead, effectively bring financial savings due the reduction of energy needed.

The term Green Software identifies application code that is designed and/or modified in such a way that its execution requires less energy compared to another, equivalent version of the same software. Usually, this is achieved exploiting techniques or implementing more efficient algorithms in terms of time of execution; less time required usually means less workload for the processor, and hence guarantee a double benefit in terms of time saving and energy absorbed. The authors at [47] have identified the following 4 macro areas to achieve better software energy efficiency in modern systems.

- Computational efficiency - To achieve computational efficiency, designers can apply coding techniques that achieve better software performance, such as efficient algorithms, multi-threading, and vectorization. In particular, keep in mind that the choice of the algorithm and data structures to be used must take in account the specific context in which the software will operate; the same choice could lead to different performance if applied in different environments with different requirements and/or goals.

- Data efficiency - Data efficiency reduces energy costs by minimizing data movement. Data efficiency can be achieved by designing software algorithms that minimize data movement, memory hierarchies that keep data close to processing elements, and application software

that efficiently uses cache memories.

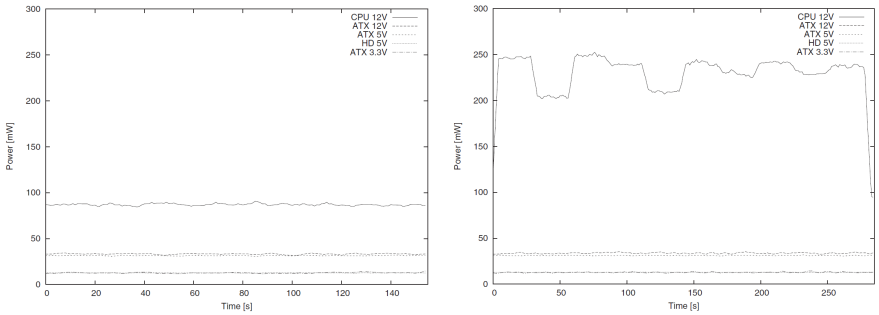
- Context awareness - The information regarding the properties of the environment in which the software is running can be used to perform particular actions, aimed especially to reduce the power demand of the system. Context awareness makes the devices "smarter" and defines the behavior of applications as passive or active.

- Idle Efficiency - Idle Power is defined as the power consumed when the system is running with software applications and services in execution, but not actively performing any workload. The challenge is to lower the idle floor by improving application idle efficiency, which will lead to a significant decrease in power needed; this is usually achieved exploiting the specific architectures of processors, and in particular the operational states in which they could be at a specific moment of time.

2.4 Memoization for green software

The technique studied in this work to achieve better energy efficiency for application software is known as memoization. This approach is based on the pre-computation of results of a given function, given a set of inputs, and the tabulation of such values in memory. The application of such technique generates a benefit when the time required to storing and fetching the result of a function from the memory is less compared to the one required to compute the value. Moreover, the energy required to access the memory is way below the one required by the processor during the computation, resulting in benefits also in terms of energy efficiency. It should also be noted that the power required from the processor depends on the load, while the other components (including memory) have an almost constant requirement in terms of power (as shown in Figure 2.3); this is partially explained by the fact that most of the power of traditional HD drives is used for spinning, and not for reading and writing operations. Similarly, dynamic RAM banks are periodically refreshed irrespective of reading and writing operations.

This concept leads to the conclusion that computational approaches that rely more on disks and memory, in place of processors, could result in lower power demands for the entire system. In general, as illustrated in the following sections, this approach guarantees more energy efficiency when dealing with computation intensive functions, executed multiple times with a relatively small domain space for the inputs. An additional advantage resides in the fact that there is no need for manipulation and/or redesign of



(a) Idle profile

(b) Under load profile

Figure 2.3: *Power consumption profiles for server hardware components. HD lines indicate power absorbed by hard disks, CPU the power absorbed by the processor, and ATX power absorbed by motherboard and other components.*

the original code, since the entire framework relies only on the intelligent use and management of memory at run time.

The term memoization was coined by Donald Michie in 1968 [36] and is derived from the Latin word *memorandum* (to be remembered), and thus carries the meaning of remembering a past result obtained from the computation of a generic function. It should not be confused with the term memorization, since memoization has a more specialized meaning in computing.

Def. *Memoization*

Memoization is a programming technique which saves in the memory of the system the result of computations of some functions, and allows them to be available for future re-uses, without a need to compute such functions again.

The basic idea of memoization is to create an n-dimensional table in which each dimension correspond to a given input for the function; for example, a function that needs two numbers as arguments will result in a bi-dimensional table. In correspondence of each input combination, the result that is obtained computing the function with the relative inputs is saved. When such function is called again, the Lookup Process will search the table and fetch the result, if available, in correspondence of the arguments with which the function is invoked; if such value exists, it is immediately returned to the caller, otherwise the function is computed and the new result

eventually saved in the table. A common example to show a typical utilization of the memoization technique is referred to the recursive implementation of the function that computes the n-th value of the Fibonacci's series. Consider the following pseudo-code which represents such algorithm:

```
int fibonacci(int n){
  if (n==1 || n==2) return 1;
  else return fibonacci(n-1) + fibonacci(n-2);
}
```

The result obtainable applying the memoization technique can be represented by the following pseudo code:

```
int fibonacci(int n){
  int result = lookupTable(n);
  if(result exists) return result;
  else{
      if (n==1 || n==2) return 1;
      else{
    result = fibonacci(n-1) + fibonacci(n-2);
    saveResult(n, result);
    return result;
      }
  }
}
```

As stated before, suppose that the function `int fibonacci(m)` could rely on a table, in the system's memory, in which is possible to save the correspondent pair `<input, result>`. The first task to perform when the function is called is to search the table for the result corresponding to the provided input; as already mentioned, this process will go under the label of lookup or `lookupTable`, and will need only the original arguments as inputs. If such value exists (and it is not an impossible result for the function), it is returned to the caller. Otherwise, the function is normally executed in order to calculate the result; just before returning the value, the `saveResult` function will save such result in the table, in correspondence of the provided input. This will be available for future computations having the same input for the Fibonacci function.

As already discussed, this approach has the advantage of not requiring a modification of the existing code, which remains exactly the same; the instructions needed to implement such solution are added (or injected) before and after the original code, leaving the latter intact. This property is highly desirable especially in situations where the source code is not available for various motivations (a common scenario in large companies), since only the compiled application (for the present work, the Java bytecode) is required.

Classical memoization approaches use large tables of precomputed re-

sults at runtime to replace potentially expensive dynamic computation. We propose to reduce the size of these tables through approximations that exploit the natural robustness of certain application domains to noisy outputs.

The approach we propose to additionally explore is approximate memoization, which extends classical memoization with the capability to interpolate previously computed and stored results to produce approximate outputs for never-before-seen inputs. Approximate memoization can enable programs to execute faster and more energy efficiently than the original program (just as with classical memoization) but with tables of precomputed results that are drastically smaller than that of classical memoization. Because approximate memoization produces approximate outputs, this approach trades accuracy of the application's results for increased performance. The goal of this project then is to produce a framework that can automatically identify and exploit computations within an application that can profitably trade off these different concerns.

```
float f_classic_memo(int input) {
  Float result = lookup(input);
  if(result!=null){ return result.floatValue(); }
  return f(input);
}
```

The function `f_classic_memo` implements a classical memoization of a function `f`. On every input to the function, the implementation first queries a table of precomputed results (indexed by the current input). If a result for this input is available, then the function returns the result; otherwise, the implementation invokes the function `f` to compute the result from scratch.

```
float f_approx_memo(int input) {
  Neighbor n = nearest_neighbor(input, DIST);
  if(n!=null){ return interpolate(n.getInput(), n.getVal(), input, linear(
      SLOPE)); }
  return f(input);
}
```

The function `f_approx_memo` implements an approximate memoization of `f` that differs from the the classical approach in that it uses a precomputed result to approximate the result of `f` for a bounded region around the input associated with the precomputed result, this stands in contrast to classical memoization, which returns a result only if the input has been seen before. The function implements this with a nearest neighbor query over the table of precomputed results to identify the closest entry (`n`) within a bounded distance (`DIST`) from the current input (`input`). Given this entry, the implementation then uses linear interpolation to produce an approximate return value. Specifically, the implementation applies a linear

interpolation of the precomputed value (`n.getVal()`) along the direction given by the difference between the current input and the stored input (`n.getInput()`). Approximate memoization produces a new computation that may produce outputs that differ from that of the original computation. The key technical challenge of this work is therefore identifying and experimentally evaluating an appropriate approximate memoization approach that improves performance of the existing application and at the same time satisfies the application's maximum tolerable error (as specified by a user).

2.4.1 Memoization related works

Memoization has been extensively studied in literature and applied to different contexts, including functional languages [41], incremental computation [25] [34] and dynamic programming [33]. The main objective of these studies was to optimize the response time of software applications, such as parsing and scientific applications, resulting in an improvement in terms of performance. The exploiting of dynamic programming regards the existence of overlapping subproblems in the process of combinatorial optimization; in this context, the reuse of a solution of a subproblem for another subproblem could reduce the overall time required. The central point in such situations resides in the determination of whether a solution can be reused or not, which depends on the context (usually an abstraction of the computation history) in which the subproblem is encountered.

Memoization can be applicable also in executing instructions; this is particularly convenient when there are operations executed, more than once, with the same operands. Similar to the proposed approach, it is possible to save the correspondent result in a table on which, through a lookup process, the following executions could be avoided (having the result already available). Sohi and Sodani [46] propose an alternative approach called Instruction Reuse, that consists in the re-utilization of the instructions in the pipeline by matching their operands. Compared to the performance objectives, little work has been proposed in terms of obtaining higher energy efficiency through power savings; among them it is possible to find the work of Azam, Franzon and Liu [8], which denotes how performance and power consumption can be improved at the cost of small precision losses in computation, exploiting instruction memoization. The approach proposed in this work is different, because it operates at a higher granularity (methods instead of instructions), although the benefits still depend on the precision of results required by the Service Level Agreement; intuitively, the greater

the precision, the larger the size of memory required to store the results, and hence the greater the time required to fetch such values.

More in general, automatic memoization has been applied in three main ways: to prevent recursive repetition of function calls, to prevent repetition over time of API functions, and to make persistent lookup tables. The first case regards recursive functions; as known, these functions recursively call themselves, changing their input parameters. A typical example can be the already mentioned Fibonacci series. The implementation of the Fibonacci function, that compute the n^{th} Fibonacci value of the series like the sum of $(n-1)^{th}$ and $(n-2)^{th}$ values, is typical of a large class of algorithms that have elegant recursive definitions. That implementation is simply unusable for most real problems because of repetition and deep stack use. An alternative is to use memoization to convert the elegant, but inefficient, algorithm into a computationally tractable one.

The second application field is that of API functions. In an interactive system, the user may invoke calculations at different times that make use the same functions with the same input parameters. In these cases, there is no central routine that can manage the calling sequence. So, it can be useful to have the routine in question that manage its own global data structure to re-use previous results; from this point of view, memoization provides an efficient, convenient, and reliable alternative.

The previous application fields show that memoization can eliminate the repeated invocation of expensive calculations. In these cases, the memo table is built during the execution of the application and perform optimization for the subsequent invocation of the function. Memoization also can be effectively applied to functions that are time-invariant with respect to the output values and when the first invocation of a function is too expensive to perform at run time. The results of these functions could be pre-calculated, saved in persistent hand crafted lookup tables, and applied in cooperation of memoization techniques. However, for temporary conditions, it would not be useful the expended effort to build the tables, in most of the cases.

2.5 Pure function definition

Memoization can not be applied to every function; methods must satisfy some properties in order to be declared as pure, and hence potential targets of memoization. Considering a function with the common formulation of 1 to 1 input-output mapping, the following definition can be proposed.

Def. *Purity*

A method is considered *pure* if it satisfies the following properties:

- it is *side-effects free*, meaning that its execution does not generate any other effects, with the exception of the original result;
- it is *deterministic*, meaning that its output depends only from the arguments passed to the method.

More in detail, the absence of side effects means that the method can only create and modify objects that are not visible outside of the method itself; every new instance of objects will be lost when the method returns the result to the caller. Moreover, the method can not access or interact with third-party components outside the environment of execution, meaning that, for example, it can not generate network traffic, nor write on the file system.

Def. *Side-effects free*

A method is *side-effects free* if the only objects that it modifies are created as part of the execution of the method itself.

The deterministic requirement for methods is equivalent to the one referred to common mathematical functions: in correspondence of the same inputs, the method should provide the same output. This property implies that the result of a function can not depend on the status, or the value, of external (global) elements and/or statuses; a method that, for example, produces different results in correspondence of the same inputs but different hours of the day, does not meet the deterministic requirement. While the concept it is easy to understand when dealing with primitive types, the issue is not trivial when considering, for example, instances of classes in Java. This leads to additional considerations that must be taken in account when dealing with such types, that have to consider not only the internal status of the referred object, but also other factors (i.e., the memory location in which they reside).

Def. *Determinism*

A method is *deterministic* if each call with equivalent inputs produces results that are identical between each other.

This work will present in the following chapter the definition of the op-

erative concept of pure function, specifically related to the correspondent Java implementation; the formulation will then be extended and modified following the proposed work, that can better delineate the property (mainly due the development of additional analysis tools). It should be noted anyway that a precise and restrictive definition of purity for a method should be always required, in order to avoid manual inspection of code and enable the applicability of such technique on a large scale.

2.5.1 Pure functions related works

From the point of view of the automatic individuation of pure functions (and hence methods that could successfully benefit from the application of memoization techniques) in the Java language, some work has also been proposed. In [51] the analysis is executed from a dynamic point of view, observing the software during its execution in the Java Virtual Machine (JVM). Different definitions of dynamic and static purity definitions are given; however, the latter results too restrictive, rejecting potentially memoizable functions due over-conservative assumptions (for example, a method can not instantiate new objects). A similar behavior is encounterable in [52], where the analysis is still dynamic and performed by a Just In Time (JIT) compiler; the same observations for the previously cited work hold. In [26] the analysis is performed only on a subset of Java functions called Joe-E. However, such approach is not feasible with the defined goals, since the aim is to consider situations where, for example, the source code is not available. Considering only such subset will be useful in terms of didactic results, but can not satisfy real companies requirements. Moreover, the majority of the cited works perform a dynamic analysis of the code; such approach would be negative in terms of energy savings, since it will require the execution of additional instructions each time that a method is executed. This is why the proposed approach will perform a static analysis, directly on the bytecode, to identify the purity of a method regardless of the inputs (possible, on the other hand, with dynamic analysis) and eventually to inject the required code to apply the memoization technique. Additional contributions of this work of thesis include further analysis (in terms of data dependencies within a method and sources of impurity) and bytecode modification, that could allow partial memoization of impure methods.

2.6 Memoization data structure

2.6.1 HashMap

A hashmap is a data structure used to implement an associative array, a structure that can map keys to values. A hashmap uses a hash function to compute an index into an array of buckets or slots, from which the correct value can be found. Ideally, the hash function should assign each possible key to a unique bucket, but this ideal situation is rarely achievable in practice (unless the hash keys are fixed; i.e. new entries are never added to the table after it is created). Instead, most hash table designs assume that hash collisions (different keys that are assigned by the hash function to the same bucket) will occur and must be accommodated in some way. In a well-dimensioned hash table, the average cost (number of instructions) for each lookup is independent of the number of elements stored in the table. A hash table also allow arbitrary insertions and deletions of key-value pairs, at constant average cost per operation.

A critical statistic for a hash table is called the load factor. This is simply the number of entries divided by the number of buckets, that is, n/k where n is the number of entries and k is the number of buckets. If the load factor is kept reasonable, the hash table should perform well, provided the hashing is good. If the load factor grows too large, the hash table will become slow, or it may fail to work (depending on the method used). The expected constant time property of a hash table assumes that the load factor is kept below some bound. For a fixed number of buckets, the time for a lookup grows with the number of entries and so does not achieve the desired constant time. A low load factor is not especially beneficial. As load factor approaches 0, the proportion of unused areas in the hash table increases, but there is not necessarily any reduction in search cost. This results in wasted memory.

Hash collisions are practically unavoidable when hashing a random subset of a large set of possible keys. For example, if 2,500 keys are hashed into a million buckets, even with a perfectly uniform random distribution, according to the birthday problem there is a 95% chance of at least two of the keys being hashed to the same slot. Therefore, most hash table implementations have some collision resolution strategy to handle such events. The most used technique to manage collisions is to use separate chaining (Fig. 2.4). In the method known as separate chaining, each bucket is independent, and has some sort of list of entries with the same index. The time for hash table operations is the time to find the bucket (which is con-

stant) plus the time for the list operation. This technique is also called open hashing or closed addressing. To keep the load factor under a certain limit, e.g. under 3/4, many table implementations expand the table when items are inserted. For example, in Java's HashMap class the default load factor threshold for table expansion is 0.75. Since buckets are usually implemented on top of a dynamic array and any constant proportion for resizing greater than 1 will keep the load factor under the desired limit, the exact choice of the constant is determined by the same space-time tradeoff as for dynamic arrays. Resizing is accompanied by a full or incremental table rehash whereby existing items are mapped to new bucket locations.

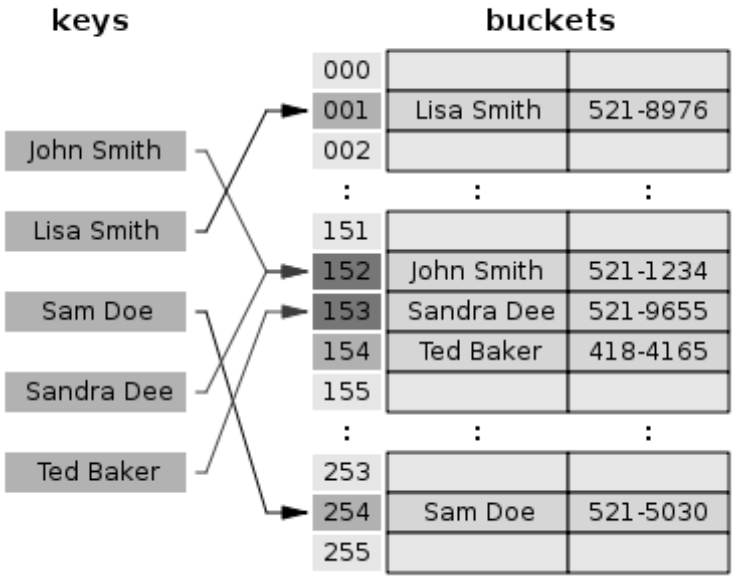


Figure 2.4: *A hashmap and the records it represents.*

The main advantage of hash tables over other table data structures is speed. This advantage is more apparent when the number of entries is large. Hash tables are particularly efficient when the maximum number of entries can be predicted in advance, so that the bucket array can be allocated once with the optimum size and never resized.

2.6.2 QuadTree

Linear quadtrees have been very popular for two-dimensional spaces. One of the major applications is in geographic information systems: linear quad-

trees have been used both in production systems. For higher dimensions, oct-trees have been used in three-dimensional graphics and robotics; in databases of three-dimensional medical images, etc... [24]. A quadtree divides terrain into four pieces, and divides those pieces into four pieces, and so on, until it reaches a certain size, and stops dividing. It splits on all (two) dimensions at each level and split key space into equal size partitions (quadrants). The Quad node structure is based on keys and the value. It adds a new node by adding to a leaf and if the leaf is already occupied, it splits until only one node per leaf.

A new vertex is inserted to a valid quadtree as follows:

- If the quadtree has no root node, place the new vertex at the root node and exit. Otherwise, start the current node at the root.
- Find the direction from the current vertex to the new vertex.
- If the current node has no child in that direction, make a new child node there with the new vertex and exit.
- Let the current node's child in that direction become the current node, and repeat.

Thus, the time to insert a vertex is bounded by the maximum depth of the quadtree. Deletion always involves removing a leaf node. It is required to set the parent's pointer to null that removes the leaf node from the tree. Technique for pruning search space: based on location of the root, we can safely ignore certain quadrants. Space subdivided repeatedly into congruent quadrants until all quadrants contain no more than one data point [43]. Searching method retrieve the matching keys and the corresponding requested values. Figure 2.5 show an example for Quadtree representation.

2.6.3 RTree

The R-tree is a height-balanced tree for indexing multi-dimensional keys. Each node is associated with a Minimum Bounding Rectangle (MBR) that encompasses the MBRs of all descendants of the node. The search operation traverses the tree to find all leaf nodes of which the MBRs overlap the query rectangle. On insertion of a new entry, the R-tree finds the leaf node that needs the least area enlargement of its MBR in order to contain the MBR of the new node. Since this data structure fits for our use case, it is detailed the search, insert, delete and intersect algorithms. A common real-world usage for an R-tree might be to store spatial objects such as restaurant locations or the polygons that typical maps are made of: streets, buildings,

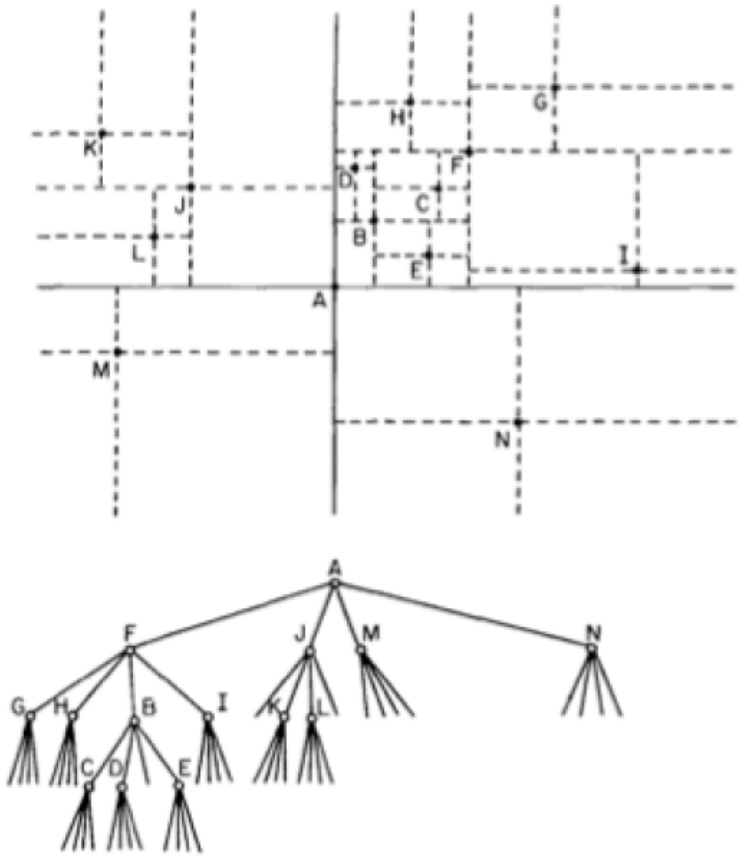


Figure 2.5: *A quadtree and the records it represents.*

outlines of lakes, coastlines, etc. and then find answers quickly to queries such as "Find all museums within 2 km of my current location", "retrieve all road segments within 2 km of my location" (to display them in a navigation system) or "find the nearest gas station" (although not taking roads into account). In order to support spatial objects in a database system several issues should be taken into consideration such as: spatial data models, indexing mechanisms, efficient query processing, cost models. Here we have an example how data takes place in R-tree and shows in two-dimensional coordinate.

There are several good reasons for the popularity of these (quad tree, interval tree, R-tree) methods, such as their simplicity, their robustness. Specifically; quad tree has a simple data structure, versatile and easy to implement, interval tree can be extended to an arbitrary amount of dimensions using the same code base and R-tree query lookup, insert and deletion

times are extremely low. Beside all the advantages, there are some reasons why quad tree and interval tree are not suitable for our case as well as their disadvantages. The disadvantage of quad tree is, if the points form sparse clouds, it takes a while to reach them because of empty spaces, space exponential in dimension d, time exponential in dimension, e.g., points on the hypercube vertices. However, quad tree divides the terrain into four pieces while a new input arrives. It is an inappropriate solution for an interval search. Though it accumulates the data in a specific area, the search structure will not allow us to take benefit from the algorithm in the best way. Moreover, Interval tree, is a massive memory footprint, which can reach as much as 16 times the size of the original data [23]. Since effective memory usage is one our primary priorities, we cannot endorse this algorithm neither. Thus, we have taken decision to use R-tree data structure for mortgage data case study that allows interval insertion in the nodes and retrieve the result in height-balanced tree in d dimension. R-tree has some disadvantages as well as advantages.

2.7 Conclusion

In this chapter, we analyze the literature related to the recent researches on energy efficiency in the IT field, also considering memoization approaches and software applications using these techniques.

There are three main open issues in the literature. First, software causes energy consumption, but the hardware infrastructure is responsible for the physical absorption of power. Thus, software energy efficiency metrics provide different results on different hardware platforms. Second, while the elementary tasks executed by hardware devices are easy to identify (e.g., operations, instructions, I/O exchanged), the same is not true for application software. In particular, the output of software depends on the input and cannot be easily standardized. To the best of our knowledge, no hardware independent benchmarks methodology has been previously proposed for software energy efficiency. Finally, the memoization approach was extensively used in order to speed up the time performance of applications, but it is never used related to the energy efficiency point-of-view. In addition, the pure function definitions are too strict to successful apply the memoization technique to reduce the energy efficiency of software applications.

CHAPTER 3

ENERGY BENCHMARKING METHODOLOGY

As discussed in Section 2.3, the role of software in determining the energy consumption of data centers cannot be neglected. Software is the first cause for energy consumption, as it drives the operations performed by the processor and, thus, influences the consumption of all the layers of a computer infrastructure. This chapter focuses on software energy consumption with the goal of providing a methodology to identify and benchmark energy efficiency.

In our approach, we estimate the energy consumption of application software independent of the infrastructure on which the software is deployed. This separation allows the definition of energy consumption benchmarks for homogenous clusters of transactions or software applications. These metrics can be used to define classes of energy efficiency within each cluster, thus enabling the comparison of the energy efficiency of similar applications/transactions.

In Section 3.1, the software energy consumption estimation de-coupling the usage of computing resources and the consumption of energy by computing resources is presented. In Section 3.2 a software energy benchmarking is defined. Section 3.3 describe the empirical testing settings. Sec-

tion 3.4.1 and 3.4.2 shown the results of the software estimation and the software benchmarking, respectively. Finally, Section 3.5 introduce a new metric in order to refine the software energy estimation using the test parameters, and presents the reduced error estimation accuracy.

3.1 Estimation of software energy consumption

The starting point of our methodology is the de-coupling of the usage of computing resources and the consumption of energy by computing resources. This separation allows us to assess the energy efficiency of software applications independent of the hardware infrastructure. In an automotive context, the consumption of resources would correspond to the number of kilometer along a route connecting two points, while the unit energy consumed by the hardware infrastructure would correspond to the liters of gasoline consumed by the car to drive one kilometer.

We model the usage of resources by a software application with three main components: the usage of CPU, the usage of the database, and the usage of the network. Our measures are always referred to the execution of a specific benchmark workload, e.g. an ERP transaction.

The total energy consumed to execute a transaction is given by the following expression:

$$E_{tot} = \int U_{(t)} * \beta'_{(u)} * dt + DB * E_{DB} + NET * E_{NET} \tag{3.1}$$

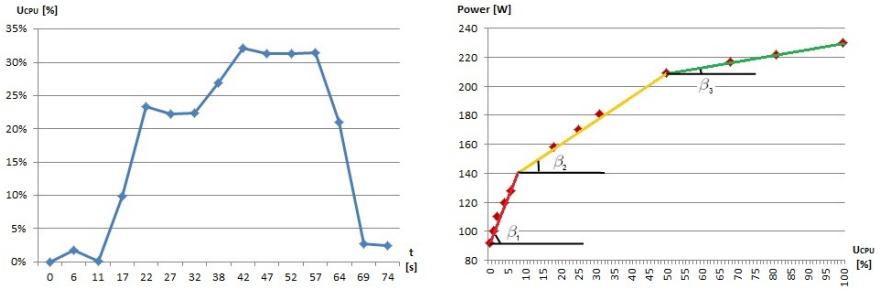
where $U_{(t)}$ is the percent usage of processor at time t, $\beta'_{(u)}$ is the first derivative of the power absorption of the CPU with respect to the percent usage (see Fig. 3.1(a)), DB is the number of bytes exchanged with the database, E_{DB} is the energy consumed to exchange 1 byte with the database, NET is the number of bytes exchanged with the network, and E_{NET} is the energy consumed to exchange 1 byte with the network.

If we assume that the power absorption of the CPU grows by linear with the percent usage of the CPU, $\beta'_{(u)}$ becomes a constant, say β', and Eq. 3.1 can be simplified with the following expression:

$$E_{tot} = \gamma * \beta + DB * E_{DB} + NET * E_{NET} \tag{3.2}$$

where $\gamma = f * \int U_{(t)} * dt$, $\beta = \frac{\beta'}{f}$ and f is the clock frequency of the CPU.

γ, DB and NET evaluate the consumption of resources and, thus, the efficiency of the application, while β, E_{DB}, and E_{NET} evaluate the unit energy consumption and, thus, the efficiency of the device. As the usage of



(a) $U_{(t)}$. (b) $\beta'_{(u)}$.

Figure 3.1: *Plot of* $U_{(t)}$ *and* $\beta'_{(u)}$.

the processor is obviously influenced by the speed of the processor itself, both γ and β are normalized by the clock frequency of the device.

If the function $\beta'_{(u)}$ is not linear, as it is often the case with modern processors, the model can be refined by approximating $\beta'_{(u)}$ as the combination of two or more linear relationships. β will be then substituted by β_1, β_2, ..., β_n (see Fig. 3.1(b)).

It is important to note that γ, DB and NET are intrinsic characteristics of the software application, while β, E_{DB}, and E_{NET} depend only on the hardware infrastructure. The two sets of metrics can be measured on different test units. Once γ, DB and NET are known for a given transaction, 3.2 allows the estimation of energy consumption on any other hardware infrastructure that has been previously profiled.

Previous work [15] has shown that the consumption of storage is almost independent of usage, as dynamic RAMs are constantly refreshed and most of the energy consumed by disks is used for spinning. Accordingly, we decided to consider the consumption of the processor only (i.e., γ and β) in the preliminary phase of our study.

3.2 Energy benchmarking methodology

The methodology described in the previous section allows us to estimate the energy consumed by a software application or, more specifically, by a software transaction. This can be applied, for example, to associate computing and energy costs with business processes that are supported by a given set of transactions. However, there are several situations in which it can be useful to evaluate the energy efficiency of different transactions in order to compare different applications with similar functional characteristics.

The γ, DB and NET metrics can be usefully employed to define benchmarks for application energy consumption, as they provide the consumption of resources independent of the hardware infrastructure. These metrics can be used to compare a sample of homogeneous transactions of the same application (e.g., transactions characterized by a high or low usage of resources, or CPU/DB intensive transactions).

Our benchmarking methodology includes the following steps:

- Define the sample of transactions to be compared;
- Classify transactions in pre-defined categories with homogeneous characteristics;
- Measure the resource usage (γ) of each transaction;
- Compute the mean value μ and the standard deviation σ of the values of γ;
- Define thresholds for energy efficiency levels based on the values $\mu-\sigma$ and $\mu+\sigma$, i.e.:
  - Highly efficient transactions: $\gamma \leq \mu - \sigma$
  - Medium efficient transactions: $\mu - \sigma < \gamma \leq \mu + \sigma$
  - Lowly efficient transactions: $\gamma > \mu + \sigma$

The same set of metrics may also be used to compare different applications that satisfy similar functional requirements. For example, a set of ERP applications could be compared by analyzing the γ, DB and NET metrics computed for a set of commonly used transactions for which similar benchmark workloads have been implemented. However, it should be noted that in corporate contexts it is often unfeasible to consider the implementation or the adoption of a different ERP system, due to the high number of legacy constraints and for compliance and business continuity issues. It may be more useful to compare the energy efficiency of different implementations of the same transaction (e.g., with different level of customization) and, in general, to assess the computing resource requirements of different business processes.

3.3 Approach to empirical testing

We have validated our methodology in a real case study. In particular, our test bed was an oil and gas company. The company is supported by a complex SAP system that enables a huge number of transactions related to all

the business processes, executed at multi-national level. We have focused on a sample of transactions that are involved in the most commonly executed processes. We have selected the three most used transactions of the SAP ECC 6 (ERP Central Component), according to the Pareto principle. These transactions support the Procurement cycle, including orders, bills and requests for purchase of services. We have selected four additional transactions non related to the procurement cycle, characterized both by an intense usage of resources and by a high frequency. The final sample includes the following transactions: These seven transactions are classified as

Table 3.1: *Classification of the benchmark transactions.*

Functional area	Transaction	Category
Procurement cycle - Purchase order	ME23N	Low resource use
Procurement cycle - Billing	MIR4	Low resource use
Procurement cycle - Payment	FBL1N	High resource use
Financial cycle	F110	High resource use
Material management cycle	MD01	High resource use
Material management cycle	ZM20_EP	High resource use
Material management cycle	ZM26_EP	High resource use

High resource use or Low resource use, depending on their use of the CPU resource (Tab. 3.1).

3.3.1 Definition of benchmark workloads

We measure the energy consumed by the execution of each transaction. This requires the definition of benchmark workloads invoke transactions automatically and stress the system. To create the benchmark test set we used VirtualUserGenerator (VuGen) within HP LoadRunner. VuGen can register, create and simulate real users' actions. Our benchmark scripts execute the SAP Logon, select the SAP test environment, perform the authentication and execute one of our pre-defined transaction workloads. Finally, they log out and exit from the system. Our scripts are executed by the web-application Performance Center of HP (HP Application Lifecycle Management framework).

3.3.2 Experimental setting

We have executed our tests on a client-server architecture with three layers: the presentation layer, constituted by the client (SAP Logon), which visualizes the results provided by the application layer; the application layer,

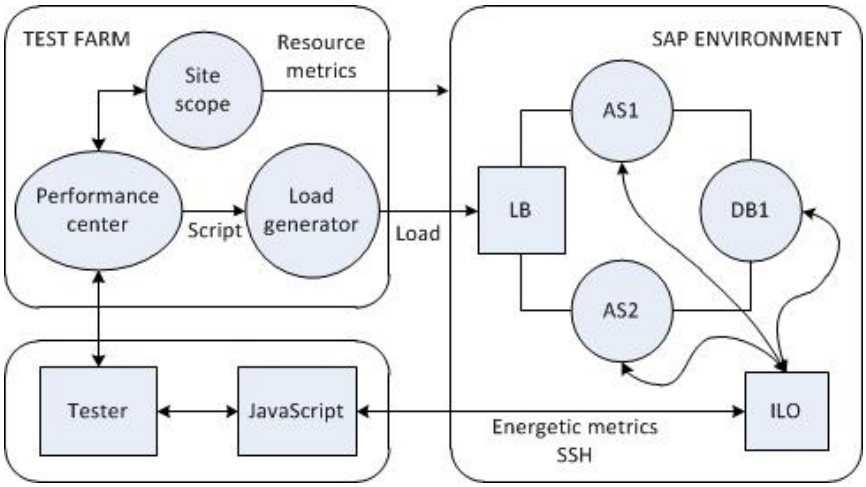


Figure 3.2: *Hardware architecture for the experimental phase.*

which implements the business logic; the data layer, based on the database server to memorize the data provided by the application layer.

We measured the usage of resources and the energy consumption of the application and data layers. The application layer is implemented on two physical servers (AS1 and AS2) with a load balancing software, while the data layer is implemented on a single physical server (DB1), as shown in Fig. 3.2. AS2 server is used only for the background execution of transactions, or when the load of AS1 is higher than a pre-defined threshold. The DBMS is Oracle 10g (version 10.2.0.4) with 222.50GB of data. The technical specifications of AS1/2 and DB1 are:

- **AS1 and AS2**, HP ProLiant BL460c G6 Server with two Intel Xeon E5520 @2.27GHz, 24GB DDR3 @1.333MHz of RAM, two SCSI 146GB HD @15.000RPM and with a SUSE Linux Enterprise Server 10 (SP3) OS;
- **DB1** is similar to AS1/2, the only differences are the two CPUs Intel Xeon E5500 @2.67GHz.

The power absorbed by the servers is measured by the HP ILO (Integrated Lights-Out), which is an embedded technology that acquires the values of power every 10 seconds. These values are transmitted by means of the SSH protocol and acquired by a script written in Java. Fig. 3.2 presents the infrastructure that we have used to measure the usage of resources in our testing environment. The load is distributed to AS1 and AS2 by the Load Balancer (LB). The measurement infrastructure is implemented by the HP

Performance Center, with the exception of the acquisition of the metrics that is performed by the SiteScope application. The controller coordinates the activities to be executed on the application servers. It also gathers and organizes the metrics provide by SiteScope. Values are acquired every 5 seconds.

3.4 Empirical testing

3.4.1 Testing the approach to the estimation of software energy consumption

We have applied our methodology to the sample of SAP transactions described in Table 3.1. We have estimated the energy consumed by each transaction in addition to the idle by means of Eq. 3.2, by considering only the processor as a first estimate. Then, we have measured actual energy consumption by means of the experimental kit described in Section 3.3.2. Tab. 3.2 shows that CPU usage is a good indicator of overall energy consumption for CPU-intensive transactions. In particular, CPU usage and energy consumption have almost identical trends for CPU usage$> 5\%$, while they show discrepancies only for very low values of CPU usage. In Figures 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, and 3.9, for each analyzed transaction, CPU and Watt trends are plotted.

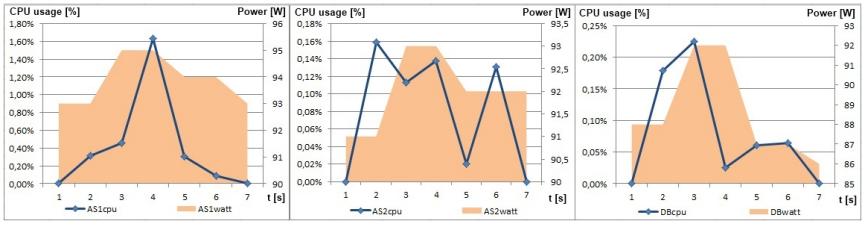


Figure 3.3: *CPU and Watt trends plotted for ME23N transaction.*

Figure 3.4: *CPU and Watt trends plotted for MIR4 transaction.*

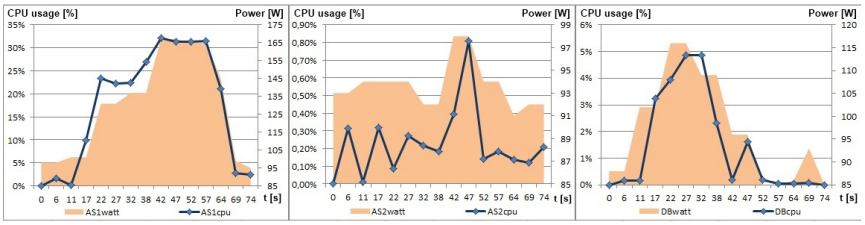


Figure 3.5: *CPU and Watt trends plotted for FBL1N transaction.*

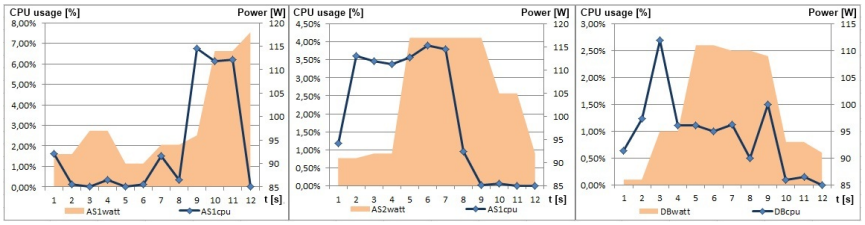


Figure 3.6: *CPU and Watt trends plotted for F110 transaction.*

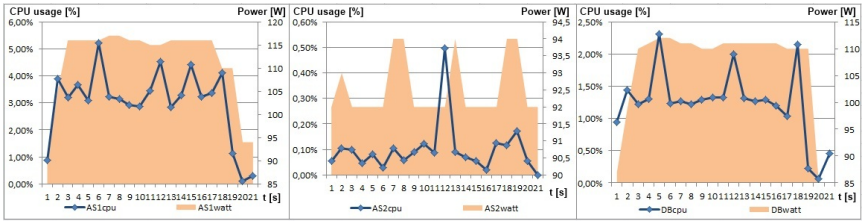


Figure 3.7: *CPU and Watt trends plotted for MD01 transaction.*

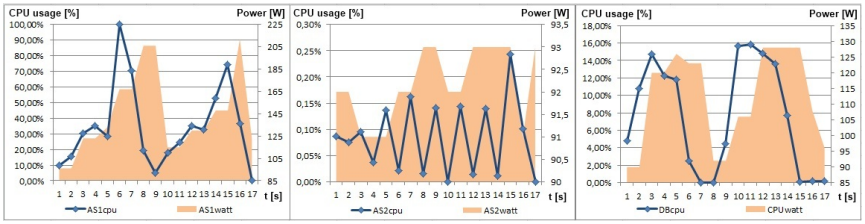


Figure 3.8: *CPU and Watt trends plotted for ZM20_EP transaction.*

Table 3.2 presents the final results of our experimental campaign. Our methodology estimates energy consumption with an average error of 2.61%. This error can be considered acceptable in a business environment. Results also confirm that the processor is the main driver of software energy consumption, while databases and networks only contribute to the idle power absorption and, thus, can be left out of the estimation methodology with a

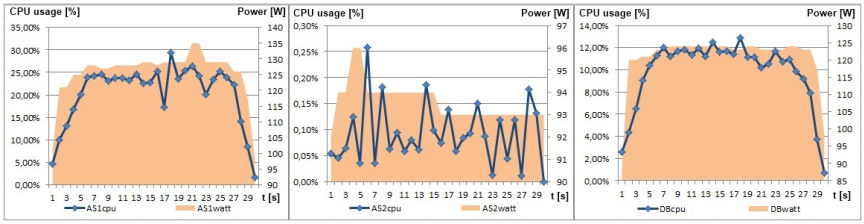


Figure 3.9: *CPU and Watt trends plotted for ZM26_EP transaction.*

Table 3.2: *Results of the validation phase. E_m is the real energy consumption measured with the ILO, E_s is the estimated energy consumption, ΔE is the gap between E_m and E_s, and $\Delta\%$ is the percent value of ΔE. Values are refer to the net energy (i.e. total minus idle).*

Transaction	$E_m[J]$	$E_s[J]$	$\Delta E[J]$	$\Delta\%[\%]$
ME23N	9,610	9,727	117	1.2
MIR4	38,825	37,316	-1,509	-3.9
FBL1N	25,625	24,951	-674	-2.6
F110	16,887	16,958	71	0.4
MD01	37,545	36,118	-1,427	-3.8
ZM20_EP	34,935	34,777	-158	-0.5
ZM26_EP	54,385	57,571	3,186	5.9

limited impact on the precision of estimates.

3.4.2 Validation of the energy benchmarking methodology

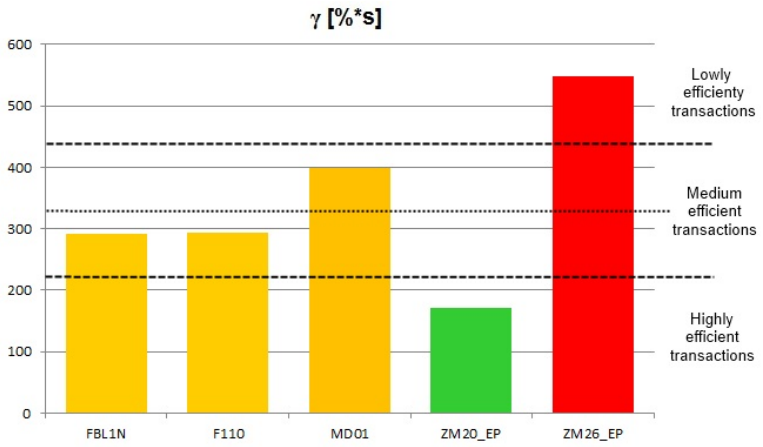


Figure 3.10: *Transactions clustered by efficiency.*

We have applied our benchmarking methodology to a sample of transactions (see Section 3.3). We have grouped the transactions into two clusters according to their perceived resource usage obtained by interviewing the system administrator. The clusters are reported in Tab. 3.1. We have classified the transactions belonging to each cluster according to the parameters $\mu - \sigma$ and $\mu + \sigma$ referred to the statistical distribution of γ values. Fig. 3.10 shows the analysis of the High resource use cluster, where each transaction is assigned to a class of efficiency.

Energy benchmarks can be useful to control the actual usage of resources of a data center along the energy efficiency dimension. Current accounting models are mainly based on data center floor space occupation, without considering the actual usage of computational resources and, thus, energy consumption of a data center. Therefore, electricity may account for up to 15% of the operating costs [29] a more accurate monitoring model may encourage significant organizational changes and allow greater efficiency. These advantages acquire even more importance in a cloud environment. The benchmarking methodology enables the comparison of functionally similar transactions and applications. This provides new metrics to evaluate software quality to select applications based on their energy efficiency, and to assess the impact of customization software energy efficiency.

3.5 Normalizing γ to generalize the benchmarking methodology

In Section 3.1, we propose a methodology to estimate the software energy consumption through a resource usage metric γ. The validation phase was executed on dedicated hardware; no validation was performed with different processor types and varying other architectural parameters. Thus, a new answer came in. Does the testbed influence the acquisition of the γ metric?

3.5.1 Experimental settings

While the previous tests are executed in an industrial scenario, in this experimental test we evaluate the ω metric in a lab setting. We execute our tests on a client-server architecture with three layers: the presentation layer, constituted by the ERP client, which visualizes the results provided by the application layer; the application layer, which implements the business logic; the data layer, including the database server storing the data provided by the application layer.

Table 3.3: *Classification of the parameters for each evaluated dimension.*

Dimension	Class Item
Number of concurrent users that executes the transaction in the same time (u)	[1; 3; 10; 20; 30; 35; 40; 45]
Transaction typology (t)	[New Business Partner; New Product; New Purchase Order; New Sales Order]
ERP application (e)	[Openbravo; Adempiere; OffBiz]
Database size (number of tuples) (d)	[0; 11; 21; 31; 41; 51; 100; 510; 10,000]
Server OS on witch the erp run (o)	[Microsoft Windows (2007 Server, 7 Home); Linux Ubuntu 10.04]
Hardware of the server (h)	[IBM X3500 Server; HP Pavillion Laptop]

We measure the usage of resources and the energy consumption of the server. The DBMS is PostgreSQL (version 8.3.5) and the servlet container is Apache Tomcat 6.0. The server's technical specifications are:

- IBM Server System X3500 with two Intel Xeon Quad-core E5450 @3.00GHz, 17GB PC2-5300 DDR2 SDRAM, 420GB HD with Microsoft Windows 2007 server enterprise and Linux Ubuntu 10.04 32-bit.

In addition, to evaluate the hardware impact on the γ metric, we execute our test on a laptop. The laptop technical specifications are:

- HP Pavillion dv6-3150el with Intel Core i5-460M @2.53GHz,4GB DDR3 SDRAM, 320GB HD with Microsoft Windows 7 Home Premium and Linux Ubuntu 10.04 32-bit.

We measure power consumption with a specific measurement kit. The current absorbed by the system is measured by means of an ammeter clamp. Ammeter clamps have a Hall current sensor inside and allow non intrusive measures. The analog signal acquired by the ammeter clamp is processed by a NI USB-6210 DAQ (Data Acquisition Board) that is interfaced via USB with a server different from the system that executes the workload. The sampling frequency employed is 250 MHz.

Definition of benchmark workloads

We focuse our tests on a sample of different transactions.

- NBP - New Business Partner;
- NP - New Product;

- NPO - New Purchase Order;
- NSO - New Sales Order;

These transactions support the Procurement cycle of three different ERPs (Openbravo, Adempiere, OffBiz).

Like in previous tests, we measure the energy consumed by the execution of each transaction. This requires the definition of benchmark workloads that invoke transactions automatically and stress the system through VirtualUserGenerator (VuGen) within HP LoadRunner.

3.5.2 Empirical analysis

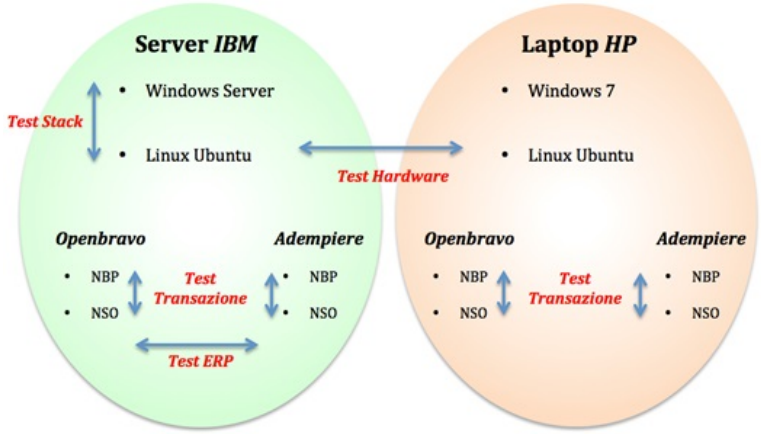


Figure 3.11: *Empirical analysis scenarios.*

Figure 3.11 shows the empirical scenarios used to evaluate the influence of the six test dimensions (see Table 3.3) on the γ metric (a.k.a. the used resources metric).

We evaluate the energy consumption of each transaction by varying each one of the six dimensions in isolation. As we will see in the following Sections, the first four dimensions (the number of concurrent users that executes the transaction in the same time unit (u); the transaction typology (t); the ERP application (e); the database population (d)) have been empirically influence the γ metric.

Scenario 1: Number of concurrent users

Table 3.4 presents the results of the first scenario. Results show that the γ metric is influenced by the variation of the number of concurrent clients.

Table 3.4: *Results of the 1^{st} scenario: Number of concurrent users u. Results for the test on IBM server with Windows Server 2007 on transaction New Business Partner of Openbravo ERP.*

User	T [s]	$\gamma[\%]$	$\gamma/user[\%]$
1	36	40.02	40.02
3	38	170.27	56.76
10	38	640.01	64.00
20	42	1,326.76	66.34
30	48	2,007.91	66.93
35	50	2,337.01	66.77
40	55	2,750.61	68.77
45	57	3,091.79	68.71

Scenario 2: Transaction typology

This scenario shows that the γ metric is influenced by the variation of the transaction typology. Similar transactions of different ERPs (New Business Partner on Openbravo, Adempiere and OffBiz) have a similar energy consumption and, thus, s similar value of γ. Instead, changing the transaction typology (e.g. high/low resource usage), has an impact on the γ value.

Scenario 3: ERP application

This scenario shows that the γ metric changes with the ERP application tested. Different ERPs (Openbravo, Adempiere and OffBiz) have different energy consumption and, thus, different values of γ.

Scenario 4: Database population

Table 3.5 presents the results of the fourth scenario. Results show that the γ metric is linearly influenced by the variation of the database population.

Scenario 5: Server OS

Table 3.6 presents the results of the fifth scenario. Results show that the γ metric is not influenced by the OS on witch the ERP is executed.

Scenario 6: Hardware

Table 3.7 presents the results of the sixth scenario. Results show that the γ metric is not influenced by the variation of the hardware on witch the ERPs are executed.

Table 3.5: *Results of the 4^{th} scenario: Database population d. Results for the test on IBM server with Windows Server 2007 on transaction New Business Partner of Openbravo ERP with 10 concurrent users.*

Tuple DB	T [s]	$\gamma[\%]$	$\gamma/user[\%]$
0	38	603.91	60.39
11	38	621.09	62.10
21	38	629.69	62.96
31	38	638.05	63.80
41	38	641.05	64.10
51	38	644.61	64.46
100	38	658.77	65.87
510	38	689.57	68.95
10,000	67	2,801.12	280.11

Table 3.6: *Results of the 5^{th} scenario: Server OS s. Results for the test on transaction New Business Partner of Openbravo ERP. Machine: IBM server. Concurrent users: 45.*

OS	T [s]	$\gamma[\%]$	$\gamma/user[\%]$
Microsoft Windows 2007 Server	57	3,091.79	68.71
Linux Ubuntu 10.04	56	3,088.94	68.64

Table 3.7: *Results of the 6^{th} scenario: Hardware h. Results for the test on transaction New Business Partner of Openbravo ERP. OS: Linux Ubuntu 10.04. Concurrent users: 45.*

Machine	T [s]	$\gamma[\%]$	$\gamma/user[\%]$
IBM server	56	3,088.94	68.64
HP Pavillion laptop	65	3,094.44	68.77

3.5.3 ω metric definition

The first four scenarios show that the first four dimensions (the number of concurrent users that executes the transaction in the same time unit (u); the transaction typology (t); the ERP application (e); the database population (d)) influence the γ metric. If we choose the transaction typology and the ERP application that we want to benchmark, we can define a new metric ω in order to normalize the energy consumption estimation. Eq. 3.2 can be modified by adding the new ω metric:

$$E_{tot}(u, d) = (\gamma + \omega(u, d)) * \beta \tag{3.3}$$

where $\omega(u, d)$ is defined as follow:

$$\omega(u, d) = [(u - 1) * p] + [(d - 1) * q] \tag{3.4}$$

where p and q are constants associated with the ERP application and the transaction typology that have been chosen for testing.

3.6 Conclusion

In this chapter, we provide a methodology to identify and to benchmark software energy efficiency. We introduce three different metrics in order to decouple hardware from software energy consumption. γ provide the resource consumption related to the software application/transaction while β define the resource usage cost. The ω metric is used to normalize the energy consumption estimation of γ. ω (defined in Eq. 3.4) is a parametric function that needs to be defined for each transaction on each ERP on witch we want to estimate the energy consumption through the constants p and q. From our results, we can further assure that the γ metric is not influenced by the hardware architecture or by the OS that have been chosen for execute the applications/transactions.

The combination of γ and ω can be used to estimate the energy consumption of applications/transactions. Thus, these metrics can be used also to define classes of energy efficiency within each cluster, thus enabling the comparison of the energy efficiency of similar applications/transactions. Benchmarks are used to identify energy inefficient software applications. In the next chapter, we propose a memoization-based approach to improve the energy efficiency of inefficient software that does not require a refactoring of code.

CHAPTER 4

GREEN MEMOIZATION APPROACH

In Chapter 3, we defined an energy benchmarking approach to estimate the energy consumption of application software independent of the infrastructure on which the software is deployed. This separation allowed the definition of energy consumption benchmarks for homogenous clusters of transactions or software applications. Benchmarks are used to identify energy inefficient software applications.

In this chapter, we propose a memoization-based approach to improve the energy efficiency of inefficient software that does not require a refactoring of code. Optimizing code has a direct beneficial impact on energy efficiency, but it requires domain knowledge and an accurate analysis of the algorithms, which may not be feasible and is always too costly to perform for large code bases. We present an approach based on dynamic memoization to increase software energy efficiency without a need for a direct optimization of existing code. We analyze code automatically to identify a subset of pure functions that can be tabulated and automatically store results.

Memoization is a programming technique to reuse computed values across a program by storing them in memory. Stored values must be indexed by the function and the input parameters that generate them. We

choose to focus on Java programs. In general, Java is not commonly used for computation intensive applications. However, this is often the case in the financial domain. Moreover, we interviewed executives at large European banks and found out that while currently only 7%-8% of the codebase of their institutions is written in Java (the larger part is still in COBOL), Java tends to be used for most of the newly developed applications. A trend can also be noted towards moving code from COBOL to Java due to the difficulty in recruiting expert COBOL programmers. We focused at the level of methods, associating the return value of a Java method with its signature and parameter values. To this end, only methods that are pure functions can be effectively memoized.

In Section 4.1, a new pure function definition is presented. In Section 4.2 introduce the SLA concept inside the memoization approach. Section 4.3 proposes a mathematical model to estimate the effectiveness of the memoization approach.

4.1 Pure Function Definition

As introduced in Section 2.5, pure functions are those functions that are *deterministic* and *side-effect free*. The need the function is side-effect free is related to the need that the execution of the function does not create any visible effect except the generation of a result. The need for the determinism characteristic is due to the fact that the function result depends only on the invocation parameters. As analyzed in Section 2.5.1, in literature there are several definition of purity. For our approach, these definitions are either too loose or too tight. Thus, we need to provide a new definition of purity to apply this concept to our approach.

From the strong purity definition, several weaker definitions can be derived that are useful in applications - a popular one in the context of the Java language is to identify as pure functions those methods that do not modify any pre-existing object, but may create new objects. For our purposes, however, such a definition is not appropriate: the newly created object must be different at each invocation, thus preventing a successful memoization. The following definition solves the above issues.

Def. *Memoizable Pure Function*

We say that a Java method m is a pure function if the following conditions hold:

- Its signature does not include object parameters, except:
  - `this` parameter;
  - array parameters, if the base type is primitive;
- Its return type is a primitive type;
- All methods invoked within m are memoizable pure functions;
- For all instructions composing the body of m, the following conditions hold:
  - The instruction does not read or modify static variables;
  - The instruction does not read or modify variables that are not declared within the function;
  - The instruction does not modify array values.

Analyzing our pure function definition we can observe that the following operations are allowed; which constitute relaxations w.r.t. typical definition of purity:

- The function can throw exceptions;
- The function can catch exception thrown by method executed within the analyzed function;
- The function can instantiate objects if their accessibility is strictly within the analyzed function.

These relaxations are strictly bound to the use of the pure functions within the memoization approach. With respect to the generation of exceptions, the operation is allowed because if an exception is thrown, the *trade-off* block will never be invocated and the passed input parameters will never be saved in the *memo-table*. Thus, for each invocation of a functions with input parameters that threw an exceptions, the lookup block will return a miss on the memo-table and the normal execution will be run. Exceptions catching is an accepted operation too. In this case, the function catches the exception thrown by another internal function and the related

operations are managed in the try-catch block. For the allocation of objects inside the analyzed functions, what is needec to maintain pure the function is that the objects will not be returned as return values. If the object visibility is strictly related to the analyzed function, when the function terminate, the object are managed by the Garbage Collector and are automatically deleted. Thus, the JVM stack and, more in general, the program state are not influenced outside the analyzed function.

Given our pure function definition, we analyze Java bytecode instructions in order to define the instructions that are not allowed to be present inside a pure Java function.

- `*astore` and `*aload`, when the arrayref reference is not related to a variable defined inside the analyzed method;
- `invokevirtual`, when the `objectref` reference is not related to an object defined inside the analyzed method;
- `invokeinterface`, because during the static analysis is impossible to get the real implementation of the method that is invocated;
- `getfield` and `putfield`, because these instruction are used to access to global variables declared outside the analyzed method;
- `getstatic` and `putstatic`, because these instructions act on static variables that are related to the class and not to the object.

These bytecode instructions must never be used within a pure Java function. In addition to these instructions, other bytecode instructions are set as unusable because for a conservative choice caused by our static analysis approach.

As an example, consider the source code of the XIRR method reported in Listing 4.1. The XIRR method computes the annualized internal rate of return of a cash flow at arbitrary points in time. The analysis is conducted on the bytecode, which is not shown for the sake of brevity. Initially, the analysis assumes the target method to be pure.

The XIRR method takes as input two arrays, respectively of double and int, representing the cash flow in terms of values and dates, and a guess of the solution used to initialize the algorithm (a double). At the bytecode level, the method signature is `([D[ID)D`, where `[D` represents an array of double. Since the array is composed of elements of primitive type, it does not cause impurity according to Definition 4.1, and the same goes for the other two parameters. Each element of the arrays will be treated as an individual parameter for memoization purposes. Lines 2-4 declare variables

Listing 4.1: *XIRR method source code.*

```
public static double Xirr(double[] value, int[] days, double guess){
  int maxConvergenza = 50;
  double adrd = 0.05, xirr = guess, sum;
  char sign = '+';
  for (int i = 1; i <= maxConvergenza; i++) {
    sum = 0;
    for (int j=0; j<value.length; j++) {
      sum += value[j] / Math.pow(1 + xirr, days[j] / 365.0);
    }
    adrd = signAdrd();
    xirr = xirr + adrd;
    if(Math.abs(sum) < 0.0000001) { break; }
  }
  return xirr;
}
```

of primitive types, and therefore do not change purity. Then, the analysis proceeds on to the two for loops. It checks the data accesses, both in read and write. The analyzed method only writes to local variables of primitive type, and accesses both array parameters only for reading. By Definition 4.1, the XIRR method is therefore pure, if we assume that `signAdrd()` is a pure function (which is the case).

4.1.1 Purity classification

During the purity analysis of the application under exam, we also evaluate different purity classes. The sources of impurities that have been identified include the following:

- the method signature;
- the return type;
- access to static variables;
- access to global variables;
- modification of object parameters;
- invocation of non pure methods;
- other sources of impurity.

Using the information obtained by the bytecode purity analysis, we also define four main characteristics that classify a method in terms of purity:

- Pure methods;

- Memoizable methods;
- Purifiable methods;
- Wrapper methods.

Pure methods A method is pure if all the sources of impurities checked are positive; if at least one of them is not verified, the method presents sources of impurities and hence can not be defined as such. Pure methods can be memoized correctly, regardless the context in which they are invoked.

Memoizable methods In particular scenarios, a method can be memoized, even if it is non pure. All the above sources of impurities should be verified, with the exception of the one related to the invocations of non pure methods. A more relaxed constraint it is allowed. With this term are identified methods that are not pure only due their return type. Moreover, it should be verified that the returned object visibility should not continue after the end of the calling method (hence, the object should not be returned again by the method under exam). If this situation is verified for every non pure method called, the calling function can still be memoizable, even if can not be defined as pure. Obviously, a pure method is also always memoizable.

Purifiable methods A method is considered solvable if it is possible to apply bytecode modifications to transform it in an equivalent function that is completely pure. Impurities that prevent the method to become solvable are the return type, writing on object parameters, the presence of non pure/non solvable calls, and the impurities grouped under the flag *other*. On the other hand, if the sources of impurities are related to the access in reading to global variables (static and non static), or to the presence of calls to wrapper methods, the original function can be transformed in a *wrapper* that calls a completely pure (and hence memoizable) equivalent method. An intermediate situation is the presence of impurities due to modification of global variables. In this case, the pure section of the method can be separated from the rest, intrinsically non pure. If the method presents such sources of impurities, it can not be defined as solvable. However, it is generically refered with the term *modifiable*, indicating that it will consists in a pure code section, followed by the set of non pure operations. The same considerations can be applied for impurities related to the modification of objects passed as parameter. Since this last case is not actually solvable, the aspect will not be considered in terms of global solvability.

Wrapper methods A method is considered a wrapper if the only operation it performs is to invoke a pure method. Given the definition, the analyzer will check the instruction list, in order to determine if no other operators outside the method call, its stack and the return statement, are present. This is the common scenario created by a resolution of a method which presents impurities due the access in reading to global variables; the wrapper method will only call the pure section of the method, providing on the stack the global variables needed, and executing no other operations. Methods that satisfy this property can be the target for bytecode modifications that allow to call directly the pure section, expanding in this way the pure code section for the calling method.

4.1.2 Differences w.r.t. traditional definition of purity

Analyzing the purity classification described above, we can state several differences between the traditional definition of purity. The relaxation of some constraints allow to expand the set of functions that can be effectively considered pure. In particular, we state the following differences:

- Since it is now possible to identify the reference of instructions such as `*astore` and `*aload`, the presence of the correspondent opcode in the instruction list does not automatically prevent the method to be pure; instead, only operators having as reference an address related to a parameter determine the non purity of the function;
- related to the previous, is now possible to have mono and multidimensional array as input and as return type. Impurities for such aspects are now related only to the presence of generic objects in the signature;
- Instructions such as `getfield` and `putfield` determine the non purity of the method only if they refer to an object passed as parameter, or to the calling object itself. Operations having as target objects defined within the function are now not a problem in terms of purity;
- `invokevirtual` instructions that refers to objects passed as parameter are not an issue, since the check has already been managed regarding the signature. However, in a future scenario in which objects are eligible as input, the check regarding the purity of the invoked method is enough. `invokeinterface` instructions that target such objects can still prevent the purity of the method; the subject could be studied in future developments, since it is necessary to identify all interfaces implemented by a class to correctly identified the referred method;

- The management of static variables has not changed, since `getstatic` and `putstatic` instructions do not require the address of the target (the information are located directly in the instructions); hence, the presence of such opcodes is enough to define the method as non pure.

4.2 Memoization Service Level Agreement (MSLA)

Service Level Agreements (SLAs) can be described as formal written contracts developed jointly by a provider of services and its users. Since SLA are living documents, the responsibilities, and expected performances are recorded. In our financial application use case, the SLA is the precision of decimals in the computed result. So, some important terms are highlighted for this concept which is accuracy on the result, SLA precision and bias.

These terms have a specific meaning when they are applied to data on which the decimals have significance, though accuracy and precision are generally interchangeable with each other in common use. Accuracy is a measure of how closure our estimate to the true result. On the other hand, SLA precision is determined by the final user that is a measurement of how close are the replicated estimates from each other. This is the same as asking how much error is there around a mean estimate. Then, bias occurs when our estimates are either systematically larger or smaller than the true value.

4.3 Memoization Performance Model

The effectiveness of our approach depends on the energy required by the function for a computation compared with the energy required to read a memorized value, and on the hit rate of the stored values [7]. In turn, the hit rate depends on the variance of the input parameters of the function and on the size of the memory available for the memoization. We developed a model that allows estimating the effectiveness of our approach taking the execution time as a proxy of energy consumption. We consider execution time because it is easier to assess the time performance of a function by means of profiler tools rather than to evaluate its energy consumption, and energy consumption is directly related to execution times for computation intensive applications.

We define memoization effectiveness as:

$$\eta = \frac{T}{T_e} \tag{4.1}$$

where T is the average time required to satisfy a function call through our approach and T_e is the time required to execute the function and compute the result. If α is the hit rate then:

$$T = \alpha T_{hit} + (1 - \alpha) T_{miss} \quad (4.2)$$

$$T_{hit} = T_m \quad (4.3)$$

$$T_{miss} = T_m + T_e + \beta T_t \quad (4.4)$$

where T_m is the time to read a memoized value that is stored in the memory, and T_t is the time required to execute the trade-off module that decides whether or not to store the new value of the function.

Initially, the trade-off module will allow to allocate the available memory to the different functions in order to maximize total energy efficiency as estimated by means of the performance model described in this section. After this initial tuning, the trade-off module is invoked with a given frequency (β) to retune the memory allocation. In fact, a continuous execution would deeply affect performances. Equation 4.2 combined with 4.3 and 4.4 leads to:

$$\eta = \frac{T_m}{T_e} + (1 - \alpha)(1 + \beta \frac{T_t}{T_e}) \quad (4.5)$$

Equation 4.5 shows that the maximum effectiveness of the memoization approach, obtainable when the trade-off module is not executed ($\beta = 0$) and when all the input values of the function are stored in the memory ($\alpha = 1$) is given by the ratio of the time to access the memory divided by the time to execute the function. As T_m and T_e can be easily measured, this result can be used to identify a priori the set of functions that are worth being memoized.

The hit rate α depends on the number of stored values on the statistical distribution of the input parameters with which the function is invoked and it influences the time required to satisfy the call. Let us suppose that we are applying our approach to a function $y = f_{(x)}$ that is invoked with a Gaussian distribution of the input parameter x, with mean μ and standard deviation σ. If x is a continuous variable, a sampling unit τ will have to be determined. The sampling unit should be decided according to the precision needed by the final users.

To understand all the components that influence the parameter α we define the number of values that can be memoized N_d as:

$$N_d = \frac{S_m}{S_d} \quad (4.6)$$

where S_m is the size of the total available memory and S_d is the size of memory necessary to store a pair (x, y). The probability that the function f is invoked with a value of the input parameter x that is tabulated is:

$$\alpha = erf(\frac{N_d * \frac{\tau}{2}}{\sigma * \sqrt{2}}) \tag{4.7}$$

where erf is the error function defined as:

$$erf(x) = \frac{2}{\pi} * \int_0^x e^{t^2} * dt) \tag{4.8}$$

Equation 4.5 combined with Equation 4.6 and Equation 4.7 allows to estimate the effectiveness of our approach for a given function with a specific variance of the input parameter σ according to the available memory S_m

4.3.1 Performance Model with MSLA

Analyzing Equation 4.5 we can say that the hit rate α is influenced by the parameter τ, because all the other parameters have fixed value. So we have to define the sampling unit τ. As it is said before, the sampling unit is decided according to the precision taken by the final users. We call that precision value SLA (Service Level Agreement) and we define it as:

$$SLA = f(\tau) \tag{4.9}$$

So τ is an inverse function of SLA, particularly is defined as:

$$\tau = f^{-1}(SLA) \tag{4.10}$$

Given some training data, it is always possible to build a function that fits exactly the data. But in the presence of limited memory usage, it is not possible to use all the given sampling units by considering the repetition of same samples which would lead to a poor performance. The general idea behind the design of a model is thus to look for a fitted function with memoization method. Typically, one would look, in a collection of possible models, for the one which fits well the data. But there is an important question: how many sampling units (τ) have to be used in order to have an accurate function? The larger the sample size, the more precise is the estimate. Sample size will therefore depend largely on the reliability wanted to place in the estimate. If the request is a very precise estimate, a larger sample of inputs is needed than if a good approximation is wanted. Statistical methods requiring the collection of some pilot data, are available

for calculating sample sizes necessary to achieve predetermined levels of precision.

To achieve precision, the idea is saturating memory with input/output data by executing the program during the tuning phase. It is needed again to control the memory if the input values are same with a new input data. In this way, the output value will be the same. Thus, in order to have good accuracy in the function, the repeated data is not saved in the memory, letting the other sampling units take place in it.

4.4 Conclusion

In this chapter, we proposed the use of memoization in Java methods to reduce the energy consumption of pure functions without a refactoring of code. To this end, we have introduced an appropriate definition of weak purity and a purity classification that allow to expand the set of functions that can be effectively considered pure. Then, Service Level Agreements are introduced in the memoization concept to introduce the accuracy of results of the modified pure functions. Finally, two model are defined in order to estimate the effectiveness of the memoization approach, taking the execution time as a proxy of energy consumption.

CHAPTER 5

GREEN MEMOIZATION SUITE (GREME)

The architecture of Green Memoization Suite (GreMe) is showed in Figure 5.1. The architecture is composed of three main components: the first is related to the static bytecode analysis (to individuate which function could be potentially memoized) and the bytecode modification, necessary to inject the required code portions; the second one deals with the execution of the meta-model `Decision maker`; the last one is related to the management of data tables into the memory. Each one of these components is further described in the following sections of the present chapter.

The static analysis examines the entire bytecode of an application, looking for functions that satisfy the *purity* requirement. The information that univocally characterize a method are saved in memory and, for the one that effectively could gain benefits, the `Bytecode modification` module is executed; it will modify the correspondent code, in order to inject the necessary instructions to execute the meta-module `Decision maker`.

During the execution of the application, the JVM executes normally the bytecode. When a modified pure function is invoked, the modified code will be loaded instead of the original one. The `Lookup` function use the `Memory Management` module in order to search inside the memory if a result exist related to the function called and its input parameters. If the

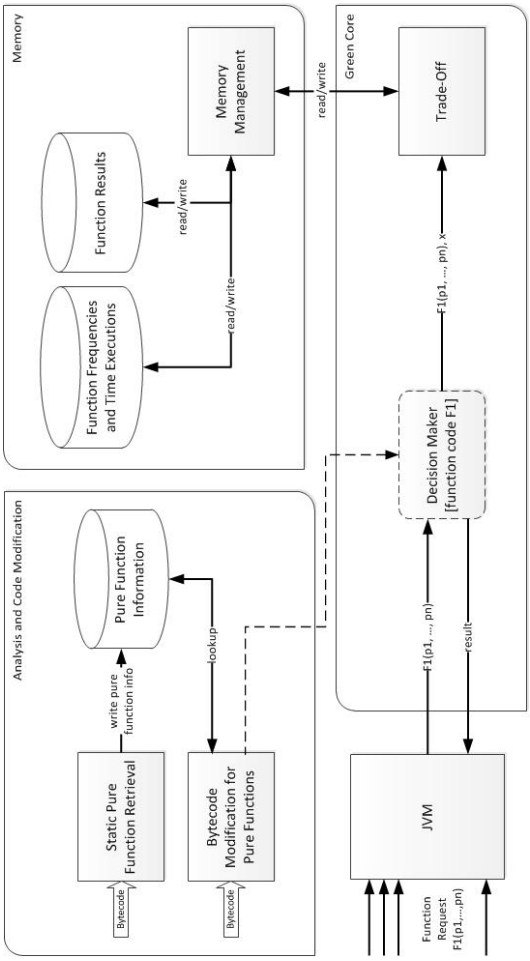


Figure 5.1: *Architecture of the existing framework*

related result exists, the function returns directly the value; otherwise, it will continue its normal execution and, just before the return statement, the module `Trade off` will be invoked. The task performed by this component is related to the possibility of saving the new result in the table, following logics related to the optimization of the memory utilization.

5.1 Execution flow of Analysis and Code Modification

The `Static pure function retrieval` module takes as input a set of compiled Java classes, analyses them, modifies them until it is possible (or useful), and returns the modified files, along with the analysis

results. This is achieved through the sequential execution of five phases, each one related to a specific task.

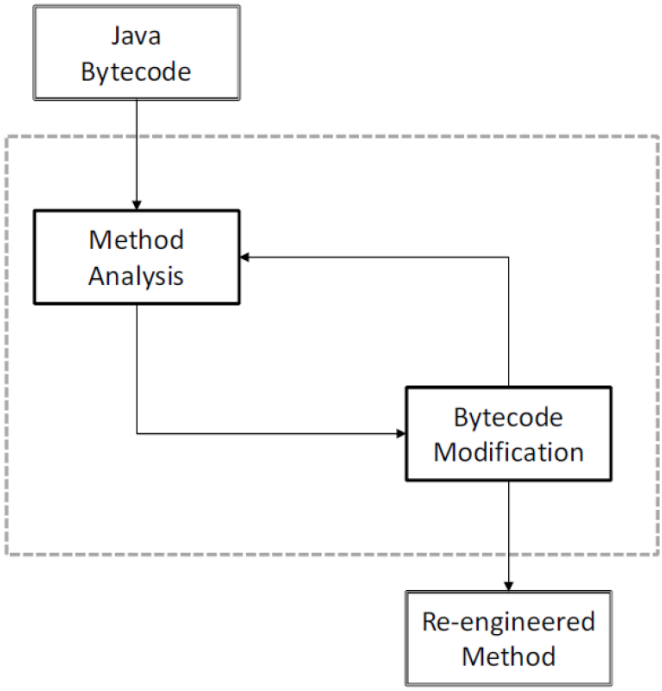


Figure 5.2: *Execution flow of the analysis and modification modules; Java bytecode is analysed and, eventually, modified, in order to obtain a re-engineered equivalent method.*

First, the application requires the path of the package to be analysed. Having the target of the operation, it builds the directories in which the results will be stored, as well as the SQL table in which the results of the first analysis will be saved. For each class, the framework will consider each method, with the exclusion of the constructors and the abstract methods (not interesting for the proposed approach). Having loaded through BCEL the necessary resources, each one of them can be univocally identified by the triple `<Classname,Methodname,MethodSign>`, and hence the application can start.

Before starting the analysis of the method itself, it is necessary to obtain the list of the functions belonging to the Java API that are called within it; this is achieved by the `TreeManager` class, which determines the entire set of methods and extracts from it the subset of APIs called. Having the full set of such functions, the application queries the table of Java API (residing in the external database) for information about its purity; if it is

already been analysed, the corresponding entry can be found in such table. Otherwise, the application performs a full analysis on it (better described in the following section) and save the results for future uses. At the end of this phase, all the information related to the purity of the Java APIs needed by the method are located in the local repository of `MethodSummaries` (see Section 5.2.1 for additional details regarding the data structure).

Having all the resources needed, the application starts with the analysis of the considered function. The tree of functions calls is generated by the `TreeManager` class (in particular, the variant with no depth for the Java API), its dimension evaluated and the relative graph plotted. Then, starting from the bottom of the tree, each method identified by a `TreeNode` is analysed. This is made from a double perspective: the first regarding the data dependencies within it (to give an overview of the behaviour of the method also in terms of its execution flow), and the latter that deals with the sources of impurities that could be (eventually) present. The results of such analysis consists in a `MethodSummary` object that (as the name suggests) summarize all the information obtained, and in the dependencies graph relative to the method. Moreover, the results in terms of purity are used to update the tree graph representation of the method, in order to give an initial, graphical overview of the location of the methods that prevent the function to be fully pure (and hence memoizable). No further analysis, at the end of this phase, can give additional information regarding the method under exam. All the results obtained are stored in a specific table, in the SQL database, that will not be further modified (it will always contain the results of the first analysis performed, with no bytecode modification executed).

Similarly to the previous phase, each node of the tree of functions calls is considered, starting from the bottom. The applications looks in the local repository for the `MethodSummary` that describes the correspondent method and, based on the information referring to its purity, can execute dedicated operations. In particular, the framework discriminates the following cases:

- the method is pure, and hence no additional operation is needed. The applications goes on with the next node in the tree;
- the method is marked as `DNM` (Do Not Modify) or it is belonging to the Java API, and hence no additional operations are possible. The first situation arises when the the user explicitly marks a method to prevent its modification (for example, functions that can be further modified, but without benefits), while the second prevents the modification of

Java API;

- the method presents impurities due the call of non pure functions within it; however, the correspondent solver has been applied and no additional modifications are available to solve the problem. Hence, any additional operation on it will be useless in terms of purity;
- the method presents impurities due the call of non pure functions within it; the `CalledMethodSolver` is started, in order to try to solve them. If the method is effectively modified, the new tree is generated and analysed again to evaluate the global impact of the modified class;
- the method presents impurities currently not solvable (in the specific, operations on arrays passed as input, or other generic impurities as illustrated in Section 5.2.2); no solvers are currently implemented to deal with such sources of impurities. The application continues its execution and considers the following element;
- the method is a `wrapper` (see definition in Section 4.1.1); no further modifications are needed for a method if this property is verified. Hence, the function is marked as `Do Not Modify` (if not already in that list) and the application continues with the normal execution;
- the method presents impurities related to the access to global variables (in reading only); this situation is dealt with the correspondent solvers (`FieldReadOnlySolver` for fields, `StaticReadSolvers` for static variables). After the modification, a new analysis is performed (as already presented);
- the method presents impurities related to the access to global variables (in writing); solvers for this situations are `StaticPuritySolver` (for static variables) and `FieldsCallerPurity` (for fields of the calling object). After the modification, a new analysis is performed (as already presented);
- the method presents impurities related to its signature and/or its return type; if even after all the possible resolutions the method is still affected by this kind of impurities, no additional operations are available to solve them. The method is flagged as `Do Not Modify`, and the execution moves over to the next element.

Once the algorithm has reached convergence, and hence no additional operations are possible (or needed), the application generates the final tree of

functions calls; it also marks all the functions belonging to it as `Do Not Modify`, in order to prevent them to be further modified (uselessly) when the algorithm will be eventually re-executed for another method.

Two last operations have to be performed before the end of the algorithm. The first one is related to the update of JAR archives which had some classes modified. In order to accomplish that, the archive is extracted in a temporary folder; then, the original classes are replaced by the modified ones, and the archive re compressed. If possible, the original JAR can be directly replaced, and the temporary folder cleaned up. The information about which archives and which classes are involved are stored in a list managed by the solvers. The last operation regards the saving of the final results in the external database; a new table is created (in order to preserve the result of the first analysis) and all the information is stored into it.

5.2 Static Pure Function Retrieval Module

The `Static pure function retrieval` module performs a static analysis of the bytecode of the application under exam. Given a set of rules to identify the purity (see Section 4.1), it determines if a method could be considered pure or not. In the first scenario, it saves the information that univocally identifies a method for subsequent studies and modifications. The user can hence decide which pure functions will be modified in the following phase. The choice is also advised through the use of additional information, such as the method main characteristics and a score that takes in consideration the number of instructions that compose the function. At the end of this phase, a list of the methods that will be effectively modified is compiled, and the `Bytecode modification` module can be executed.

The architecture of the `Pure function retrieval` module, as shown in Figure 5.3, is composed of four main components: bytecode analysis, bytecode modification, data management and main core. The present section describes the main characteristics of these elements, while each one of them will be presented more in detail in the next sections.

`bytecodeAnalysis` includes all the components related to the static analysis of the bytecode. In particular, it provides the `TreeManager` class to create and manage the tree of functions calls, the `Analyzer` class to explore the data dependencies within a method, and the `PureFunctionAnalyzer` to identify every cause of impurity that a function could present. The entire component permits to obtain a full analysis of the selected method, and provides the necessary results for the following phase.

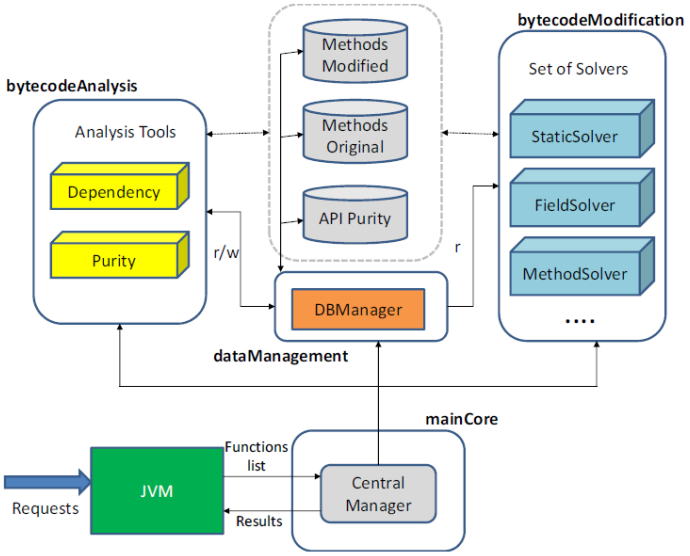


Figure 5.3: *Architecture of the* `Pure function retrieval` *module.*

As explained later, it interacts directly with the data management, in order to guarantee the integrity of the data that resides on the external database; this also prevents the analysis to be re-evaluated from the beginning, if the results are already available (following a past analysis). It also provides the methods to correctly deal with Java Archives (JAR) files (extracting, update, re-jaring).

`bytecodeModification` includes the necessary components to effectively modify a compiled Java class. For a subset of the impurities defined (and evaluated) the correspondent solver has been implemented to generate an equivalent version of the method under exam, with better characteristics in terms of purity. In particular, the component can completely solve impurities related to the reading of global variables (static and non static), plus the calls to methods that have to the property to be `wrappers` (see Section 4.1.1 for the definition); it also can partially solve impurities related to the modification of variables defined outside of the scope of the considered method. In this case, the new method will consist in a pure section (ideally covering the largest part of the code, and hence that could effectively benefit from the memoization), and an intrinsically non pure part that manages the update of such values.

The `dataManagement` component provides the necessary methods to interact with the external SQL database. It includes the functions to connect, create new tables, query and save the results obtained from the

framework. It could also be expanded with the implementation of specific queries, in order to make available additional results, such as the occurrences of each kind of impurity for a given package, the average number of methods solvable in pure equivalents, and other kind of statistics.

The `mainCore` component is responsible for the management and the coordination of the elements that compose the framework. It receives the requests related to the set of functions to be analyzed and manages all the processes for the analysis and (eventual) re-engineering of such methods.

5.2.1 Data Dependency Analysis

This section focuses on the data dependencies graph and on the role it covers in the present work. The representation of a Java class in this form can be very useful for a number of reasons related to the purity of a method, and hence to its potential memoization. One of the main challenge in this process is to build a dependencies graph from raw bytecode, without having the source code for the selected method; the present work address this problem and provide a complete and customizable graph for a generic Java method. The next sections are structured as follows: Section 5.2.1 describes in general terms the data dependencies graph, analyzing the main dependencies that can arises when analyzing portion of code and its main applications. Section 5.2.1 gives an overview of how the dependencies graph could be useful for the present work, in particular from the point of view of purity for a method, and memoization of a subsets of instructions that compose it. Section 5.2.1 presents the developed work to obtain the dependencies graph from a compiled Java class.

Dependencies graph description

The data dependencies graph is a directed graph that describes the list of data that has to be ready (or available) in order to generate (or evaluate) a given resource and/or operation. More formally, given a set of objects S and a transitive relation $R = SxS$, with *(a,b) belonging to R* modeling a dependency that can be represented textually by *"a needs b to be evaluated first"*, the dependencies graph is a graph $G = (S,T)$ *with* T *subset of* R and R being the *transitive closure of T*.

In computer science two of the most important types of dependency are data and control dependencies, with general reference towards the list of statements (instructions) that compose a program. A general *data dependency* arises from two instructions which access or modify the same resource;

since one of them has to be executed first, the latter is flow dependent with respect to the former. In other words, the second statement has to wait until the first one is fully executed, and only after that it can operate on the data it needs (which could be eventually been modified by the previous instruction). Data dependencies can be further classified under other sub-types, but for the work presented the above definition is fully sufficient.
A *control dependency* is verified when the execution of an instruction depends on the result (or the evaluation) of another instruction executed before; the most common situation is a statement representing a condition and another instruction executed only if such condition is true or false. More formally, a statement S2 is control dependent on another statement S1 if and only if the execution of S2 is conditionally determined by S1. A statement is not control dependent with respect to any other instruction if its execution is always determined irrespective of the outcome of all the instructions that preceed it.

Typical uses of data dependencies graphs in compiler technology include the instruction scheduling (algorithms such *As Soon As Possible* and *As Late As Possible* rely on this model to determine respectively the sooner and the later temporal istant in which the instruction could be executed), or for the elimination of dead code (when a variable is not usefull anymore, because no other resources actually read or write to it, it can be removed safely). More in general, data dependencies graphs found wide applications in many other different fields than the one of interest for this work; for example, in manufacturing factories, the raw materials are processed through series of intermediate stages, each one that can be modeled as a node of the graph; with a pipeline-like view, the next stage is not accessible nor workable until the previous one has been reached and completed.

Practical uses of the dependencies graph

The data dependencies graph can be useful in various scenarios when dealing with functions that violate the purity constraints defined in Section 5.2.2. The resulting graph can be personalized in a manner that underlines where and what are the impurities in the analyzed code, for example coloring in red the parts that affect the purity of the method. Considerations about them include:

- the presence of static variables can be easily spotted since each node representing a static variable has a key that starts with the letter `s`; incoming arcs mean that its state is actually modified by some other

variable (source of the dependency), and hence a side effect is produced. Outcoming arcs indicate that its value is being used by some other entity (target of the dependency), and hence the value returned by the method can be influenced not only by the parameters that it has as input, but also from some external value not retrievable in the lookup process;

- object parameters are also very easy to observe, since they correspond to nodes having key starting with `p` and a suffix containing `object`. The same observation about dependencies to and from static variables can be applied for these entities, which are generally not supported by memoization (unless setting some kind of serialized version of the object, which will anyway results in an explosion of the table dimensions). Outcoming arcs from this kind of nodes generally refer to method invocations or access to a field of the object;
- in a similar fashion, the nodes corresponding to return values (which have suffix `return_value`) can also be analyzed from two points of view. First of all, it is immediately clear if the return value belongs to a non-primitive class (matching the signature of the method), and hence needs further considerations about its life-range in the calling method; secondly, analyzing the graph backwards helps to understand which subset of parameters effectively influence the return value. This can be useful when dealing with functions having lots of input (that will usually be translated in a dimensional prohibitive matching table), but not all of them define the returned value; the ones not involved in the process can be, from this point of view, discarded, resulting in a feasible application of memoization;
- when analyzing non-static methods, side effects on the calling object are easily spotted just observing the incoming arcs for the node representing it (`p0_this_object` node, since it represent the implicit object passed to the non-static method). Outgoing arcs from that node are also signals of impurities, due the fact that represent reading access to its field variables;
- since all the invoked methods will appear somewhere in the graph, the impure ones can be highlighted (for example coloring the correspondent arc in red). If no action is taken about their presence, the fragments of instructions including them can not be considered pure.

Moreover, the data dependencies graph is a very powerful tool when deciding to memoize only a part of the method (for example, because no

actions are feasible to obtain a completely pure version of the method). This task can be accomplished selecting a subset of the graph, in which the incoming arcs will be the parameters of the new method, and the outgoing arcs the returned values (they can potentially be more than one, because the memoization allows multiple outputs). Clearly, the number of inputs will be crucial, since it will define the dimension of the lookup table (along with the distribution of their values). After selecting the desired subgraph, the new method can be obtained by wrapping up the instructions included in the considered range, and calling it in the original one (after building for it the instruction stack required for the invocation). The main problem in this algorithm is the enumeration of all possible subgraph for a given graph; at the time of writing no algorithm that solves it in polynomial time is available, although some works (such as [12]) claim that is possible to reduce it to a quasi-polynomial time, or even polynomial (with not-so-low powers anyway). It could be useful to reduce this problem to a smaller version, with the introduction of some constraints that will help to identify better candidates in terms of purity.

Building the dependencies graph: algorithm and implementation

This section shows how the problem of obtaining a data dependencies graph from a compiled Java class, and hence from the list of bytecode instructions that compose it, has been addressed for the purposes explained in the previous paragraph. Since at the moment of writing no tool is (freely) available on the Internet, at our knowledge, the decision to build an utility from scratch for the task has been taken. In fact, no external library has been used with the exception of the ByteCode Engineering Library (BCEL) [21], probably the most complete to interact at various levels with the raw bytecode. An example of generic data dependencies graph obtainable from the application of the algorithm is reported in Figure 5.4. The graph is created starting from the function `function_F`. During its execution, `function_F` invokes both the `sin` and `method_G` functions; the latter also invokes the `sin` function within its code. In Java, the mathematical method `sin` is implemented as the call to a `native sin` function, which performs the effective computation.

The proposed algorithm The algorithm for generating the data dependencies graph is different from the well-known related algorithm that will be useful for the analysis related to the purity of function. Basically, the algorithm

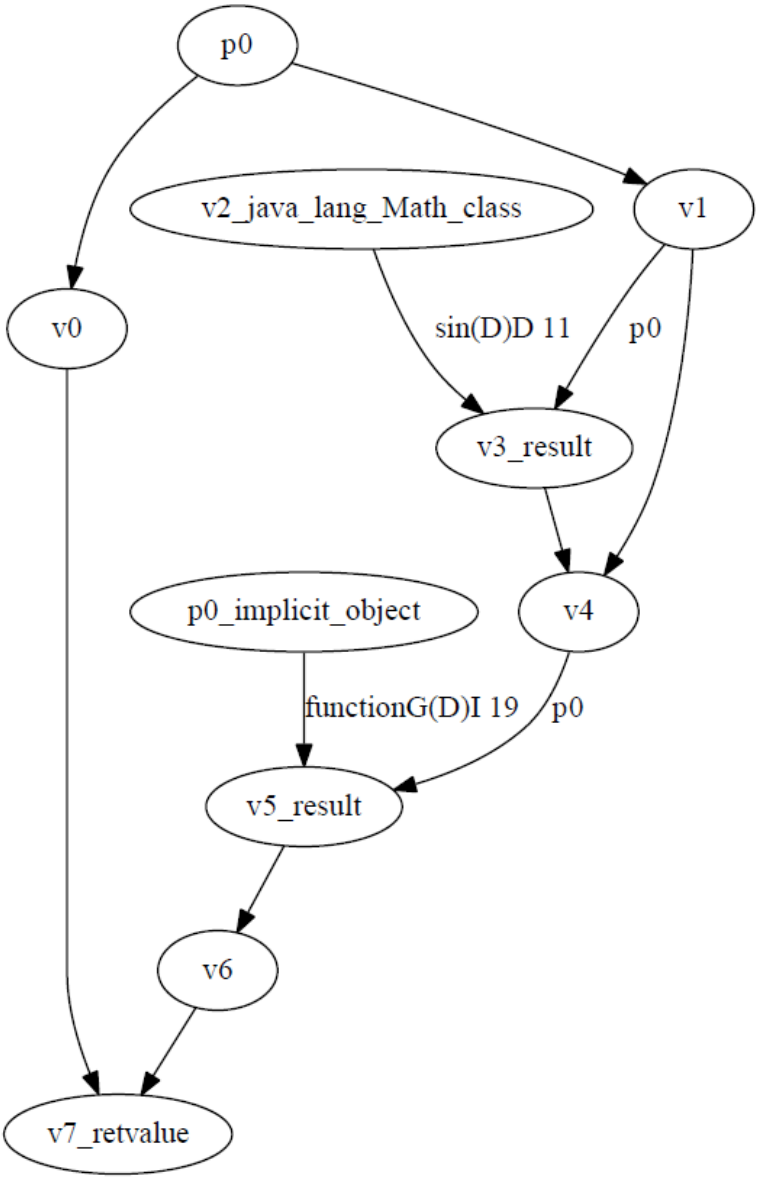


Figure 5.4: *Example of generic data dependencies graph.*

is structured in four consecutive steps (initialization, identification of the nodes, analysis of the dependencies, finalization), plus the set up phase. The only inputs required for the correct utilization of the tool are the full path to the compiled Java class, the name of the method subject of the analysis, and a the method signature (which constitute the set of information

Data: Method name, Class name, Method signature
Result: Dependencies graph for the method

Initialization;
for *all bytecode instructions in the method* **do**
  Node identification;
end
for *all identified nodes* **do**
  Dependencies identification;
end
Finalization;

Algorithm 1: Data dependencies graph realization algorithm.

required to univocally identify a Java method).

Data structures Various data structures are set up and used by the analyzer in the process. In the following are described the most important and significative classes used within the process.

- `Dnode` - It identifies a node in the dependencies graph, and hence represents a generic variable in the correspondent source code;
- `Darc` - It identifies a connection (with direction) between two nodes in the graph;
- `BasicBlock` - Used as the elementary entity in the control flow graph;
- `Btree` - Each node of the Btree corresponds to a bytecode instruction;
- `MethodSummary` - Used to wrap up all the results obtained from the analysis of a method.

Phase 1 - Initialization Three different operations are carried out in this phase of the algorithm, each one resulting in a new resource that will be used to correctly generate the full graph. The first one is the Control Flow Graph (CFG). It describes how the execution of a code fragment can be influenced by jumps corresponding to branch instructions. A basic block is a portion of code that has the important characteristic of having a single entry-point and a single exit-point. In other words, there is no way that the execution jumps to somewhere in the middle of the basic block code. The first

instruction of the block is always considered first, and each one of the following, until the end of the block, will be executed sequentially. Usually, a block ends in correspondence of a jump, a branch instruction or a return statement, while starting at the location of an instructions being target of jumps or "fall-through" instructions following some conditional branches. Given these definition, it can be stated that a block is characterized by the set of the blocks whom execution preceed it (inputs), the set of the blocks whom execution follow it (outputs), and the first and last instruction of the relative code portion (boundaries). In order to generate the correct sequence of basic blocks, the algorithm used to build the CFG identifies all the boundaries, each one with the right boundary. Starting from the first instruction, each one of the following is analyzed; the end of the block is always in correspondence of conditional branches, uncondinational jumps or return statements. Depending on the kind of instruction, there can be an arbitrary number of instruction behaving as entry point for other blocks. In particular:

- a return statement means that the execution will terminate after that block;
- an unconditional jump identifies the entry point of the following basic block, as the instruction targeted by the jump; moreover, the instruction just preceding the target will be, by definition, the end of another basic block;
- a conditional branch identifies entry points for two additional blocks: one as described in the previous case, and another one given by the normal fall-through of execution (the instruction immediately following);
- the switch instruction is an extension of the previous case, which can add an arbitrary number of blocks (with respect to the number of cases it provides) to the graph.

Having the complete list of boundaries (entry points and end points), and hence the full set of blocks that generate the graph, the last step is to compute for each one of them the set of inputs and outputs. Usually, only one set is computed, while the other one is obtained by inverse procedure; the easiest way is to compute, for each block, the set of its outputs, considering just its last instruction. The end point, in fact, provides all the information needed to understand which and how many blocks can follow the object of the analysis during the execution. As last task, for completeness, all the

inputs are also computed. Now the graph is fully described and hence complete; given an instruction anywhere in the code, the information about its position can be used to understand at which basic block does the instruction belong. The function `printCFG` can be used in order to obtain a textual representation of the control flow graph; an additional functionality could be implemented to generate a Graphviz-compatible file to provide also a graphical representation. The second step in the set up phase is the initialization of the graph with the nodes referring to the parameters that the examined method takes as inputs. The implemented function responsible for the operation is `initGraph`, which has to take as input also the set of types referred to the parameters. This information is used to determine the starting memory location for each one of them. In fact, they are stored sequentially starting from location 0 and each one of them has dimension exactly equal to 1 (for characters and numbers, the correspondent register contains the value of the variable; for arrays and objects in general, it contains the memory address in which they reside). The only exception is the textttdouble format, which occupies two consecutive locations. In the case of having a method with signature `IDI/V` (parameter 1 and 3 `integer`, parameter 2 `double`, return type `void`), the first parameter is allocated in memory location 0, the second one (`double`) in memory location 1, and the third one in memory location 3 (since the `double` type requires two locations for itself).

The result of the operation is a set of nodes (each one having the name in the format `p+incremental number`) representing all the parameters given to the analyzed method. For each one of them the information regarding the bytecode position in the method is set to -1 (indicating and invalid location); if the parameter is an address value, the node name will also terminate with the suffix `_object` (this will be useful in phase 3 when dealing with the dependencies).

The last operation to be performed in this phase is the generation of the branch structure. All the instruction list is examined looking for every type of conditional operation. In correspondence of each one of them, a new node representing the condition is allocated. The name format for them is `cond+sequential number`, each one of them have the correct bytecode position and the `start` and `end` field working as boundaries for the condition validity; moreover, for `if-else` branches, the information regarding the bytecode position of the `else` branch is also saved in the node. Since for this kind of nodes is not correct to refer to a memory location, the correspondent field is set to -1, in a similar fashion of what happens to the method parameters. This operation could also be moved in

phase 2 but, due the fact that it not introduces additional complexity (other than another full scan of the instruction list), for simplicity reasons it is preferable to execute it before dealing with the rest of the algorithm.

Phase 2 - Identification of the nodes The second phase of the algorithm, as the name indicates, is responsible for the allocation of all nodes composing the graph, and more in general of the individuation of all the operations that generate data dependencies. The instruction list is examined sequentially, and in correspondence of particular instruction types the relative operations are performed. The following paragraphs explain in detail the behavior of the algorithm, regarding the different sets of instructions examined.

Assignment - creation of new variables These operations are used every time that a new variable (of any type) is allocated, or whom value is updated. Each one of these instructions is translated in a new node in the graph, having the bytecode position information correspondent to the instruction one. The name format for these nodes is `v+sequential number`, with an additional suffix of `_object` if the instruction belongs to the `ASTORE` class (and hence the variable is representing a memory address). Two additional information are added in this process: the local memory location which the instruction refers, and the number of instructions which generates the relative expression. A particular case is given by the bytecode instructions `IINC` (opcode 132), which increments a given integer variable by another integer value; like the previous cases, this reflects in an additional node on the dependencies graph, but, due to its particular structure, the number of additional instructions required for its correct execution is 0. To better understand the difference, in terms of bytecode, that this last operation presents with respect to the others, refer to Table 5.1.

Table 5.1: *Differences between the bytecode resulting from normal assignment of variables and the `IINC` operator; the first includes the standard stack of operators, while the latter uses only a single instruction, with the relative parameters.*

```
0  bipush 10
2  istore_1
3  iconst_3
4  istore_2
5  iload_1
6  iconst_5
7  iadd
8  istore_3
9  iinc 2 by 5
12 return
```

```
int a = 10;
int b = 3;
int c;

c = a + 5;
b = b + 5;
```

Conditional branches As previously explained, the nodes referring to conditional branches has been already generated and added to the graph. In this phase, the information regarding the number of instructions that generates the conditional expression is added; no additional changes are made. It is also possible to know the type of branch that the instruction generates, considering the parameters provided to it; this can be made in an automatic way using the implemented function `identifyBranchType`.

Assignments on objects public field This instruction does not requires the creation of a new node in the graph, but has nonetheless to be saved and used later when dealing with dependencies. For this kind of operation the usual information about bytecode position, memory location and instruction stack depth are saved in a separate structure. The difference from other nodes resides in the local memory location; the information is not carried by the examined instructions (like normal `stores`), but resides in the `ALOAD` instruction just before the beginning of the expression (as better shown in Table 5.2). An additional check has to be made in case that the referenced object is a static one; in this scenario, the memory location is not available and hence the name of the static variable has to be saved instead.

Table 5.2: *Bytecode representation of the `PUTFIELD` operator; such instruction includes only the reference of the field (in the example, `x`), while the memory location of the object modified (`n`) is carried by the relative `ALOAD` statement.*

1 aload_1 2 bipush 10 4 putfield #42 <Dnode/x I>	n.x = 10;

Assignments on elements of arrays The behavior of instructions belonging to this kind of class is similar to the one described in the previous paragraph. In a similar fashion, no new nodes are added to the dependencies graph, but the information are saved anyway in a separate structure. In this case, the examined instruction is preceded first by an expression representing the value to be assigned; this block of instructions is preceded by another expression, indicating the index on which the operation is performed, and the `ALOAD` instruction that points the structure which is going to be affected (refer to Table 5.3 for an example of the relative bytecode). Multiple copies of `index-reference` blocks are present in the event that the structure is multidimensional; only the last `ALOAD` instruction is useful to gain the memory location of the structure (the `AALOAD` operator, in fact, does not

carry any information about that).The same attention must be used when dealing with a static structure; in this case, the operations are the same as described in the current paragraph, with a `GETSTATIC` statement to determine the location of the interested variable (since it is literal and not numerical).

Table 5.3: *Bytecode representation of a generic `store` in an array; the memory location of the array modified is carried by the relative `ALOAD` statement.*

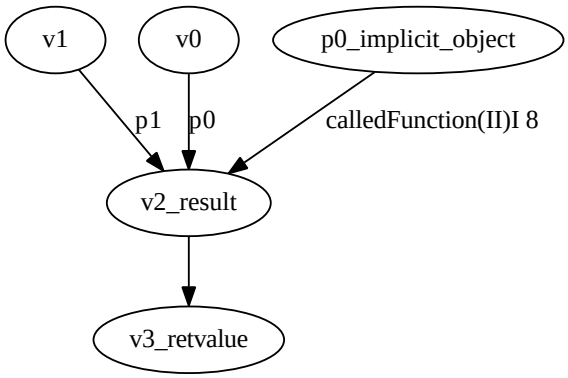
1 aload_1 2 iconst_3 3 bipush 8 5 iastore	a[3] = 8;

Static variables The management of static variables presents some differencies. During this phase, each time a `PUTSTATIC` instruction is encountered, a new node is added to the graph; the format of the key used in the hash map is `s+incremental counter` (where the counter is different from the one used for non-static variables), while the name of the node itself corresponds to the name of the static variable (which can be obtained from the `Constant Pool` of the class). The node will have the usual information about bytecode position and number of instructions needed for the expression, while the memory location is set to an invalid value, because it has no sense for this kind of variables. In the case that a `GETSTATIC` instruction is encountered before any other `PUTSTATIC` that refers to the same variable, a node that represents the variable must be added to the graph. This new entity will have exactly the same characteristics described before, with the exception of the number of instructions needed for the expression (this can be seen as a parameter given as input to the method).

Return instructions Each return statement of the method, except in the case of return type equal to `void`, will result in a new node in the dependencies graph. It will have the bytecode position as well as the `instruction stack depth` information (like a normal `STORE` instruction), but its local memory location will be set to an invalid value, since it is meaningless. Moreover, to underline the fact that the node corresponds to a return value, its name will have the suffix `_retvalue`; hence, by construction, every node in the graph with that suffix will correspond to a node without any arc pointing out (there could not be any node which depends from a return value).

Invoke instructions When encountering a call to another method, the first task performed from the analyzer is a local research to determine if the called method has been already examined or not; in this second case, a the basic information that describe it are retrived from the external database (in such event, not all the described information for the `MethodSummary` could result available). After that, the analyzer looks for the caller of the method; that can be an object (identified by an `ALOAD` instruction, in the case of a non-static method) or a class (if the method is static, and hence referred by an `INVOKESTATIC` instruction). The information is saved, and in the case of an `INVOKESTATIC` instruction, the node corresponding to the calling class is instantiated in the graph (if not already present). After that, the analyzer creates the node corresponding to the method result in the graph; its name follows the already mentioned format, with the additional suffix of `_result`. Moreover, in its notes, a list of the parameters that effectively influence the return value of the method is created (the information is available in the `MethodSummary` object that describes the considered method). An example of the obtained representation is shown in Figure 5.5 and the related code fragment.

Figure 5.5: *Dependencies graph generated from the `call` function. In this example, the method `calledFunction` is invoked with the variables `a,b` (respectively `v0,v1` on the graph), while the function is called from the implicit object. It produces a result, that will be then returned by the original method.*




```
public int call(){
  int a = 3;
  int b = 8;
  return calledFunction(a,b);
}
```

The last operation to be performed regards the eventuality that the called method modify also objects passed to it as parameters (being them static or non-static). If so, a new node corresponding to the modified instance of the object is added to the graph; in the final result will be clear to which method (and more specifically to which parameter) the new node refers. The effective dependencies that the object undergoes can be retrieved in the analysis result of the called method. The combination between the two results permits to define exactly which are the dependencies related to such entity.

Obtaining the local memory location from a load-store instruction With a few exceptions, each node in the dependencies graph has an information regarding its memory location. This will be very useful in the next phase of the algorithm, in order to obtain the dependencies from a node that consider previous values in its expression (due to the potentially different flow of execution). The function, dedicated to the process of gaining the local memory location information from the considered instruction, is available in class `DataDependency`, under the name of `getMemLoc`. Basically, it analyzes the `LOAD` or `STORE` instruction and returns the index of the value considered; in the case of an immediate instruction (such as `iload_0` or `istore_2`), the desired number is the one just following the underscore, while for normal `LOADs` and `STOREs` it is indicated by the parameter following the instruction (for example, `istore 8`); for the particular case of the `IINC` function, the register number is the first parameter (while the second is the size of the increment). The function returns -1 if the instruction passed as parameter does not provide a valid memory index.

Finding the instruction stack depth for a node One of the most relevant tasks for this phase is the computation of the number of instructions needed by each node to fully compose the relative expression. This information will be needed in phase 3, since it identifies univocally, along with the bytecode position of the variable, the entire set of instructions which could determine a dependency from other nodes; the only type of dependency that will not be evaluated using this information is the one regarding conditions, which will be managed with another methodology in the following phase. The function implemented for the task is `getVarDep`, included in the `DataDependency` class. The goal is accomplished examining one by one, in inverse order, the instructions preceding the node bytecode position, and by the realization of a pseudo binary tree, whose leaves can represent either a constant or a reference to another variable; it can also be

said that these instruction compose the set of *immediate instructions* (e.g. `*const_*` bytecode instructions), since each instruction belonging to the set does not require any additional statement to be fully specified. In the case of an expression given by an immediate instruction, the stack depth for the node will be equal to 1. Other possible situations that can be found when analyzing an expression includes:

- **binary operations**: the set regroups basically every mathematical and logical operator that requires two operands; in terms of the binary tree, an instruction belonging to the set will increment by one the depth of the tree, and hence looking forward to two more immediate elements;
- **loading from array**: this operation requires two additional information in order to be completed, the index and the reference to the structure, and hence could be considered in some way as a binary operator. The index expression is obtained by a recursive call of the `getVarDep` function, while the reference of the structure is given by an `ALOAD` instruction. The function also addresses the case of multidimensional arrays, looking for another index-expression each time an `AALOAD` operation is encountered, instead of an `ALOAD` one. In the binary tree view, a load from an array can be seen as a parent node having the left child as index and the right one as the memory reference (for the multidimensional case, just consider the last pair of operands);
- **allocation of new array**: this case consists in the analysis of a number of expressions given by the number of dimensions of the new structure. This is obtained by extracting this last information from the instruction considered and calling a correspondent number of times the `getVarDep` function;
- **unary operations**: the set includes every instruction that needs just one additional statement in order to be completed. From the point of view of the pseudo binary tree, these operations are transparent, in the sense that the analysis will not increase the tree depths. In fact, the function will still look for an additional regular expression in order to complete the branch;
- **invoke instructions**: when dealing with an invoke function instruction, the first thing is to extract from it the number of parameters that the method will need as input. These correspond to a set of regular expressions, each one to be analyzed with the usual `getVarDep` function. This operation is enough for the `INVOKESTATIC` instructions;

in the case of examining a `INVOKEVIRTUAL` instruction instead, an additional instruction is needed, in order to get the memory reference of the object calling the method (in the static case this parameter is implicit and thus there is no need of an additional instruction for it). The recursive call of the `getVarDep` function ensures that the expression is correctly evaluated, despite any number of consecutive method calls in an expression. Another particular case involves the `INVOKESPECIAL` instruction, when it refers to a new object initialization; in this scenario, after having examined all the parameters for the method, the analysis must continue until the encounter of a `NEW` instruction (usually there could be some `DUP` instructions between the two parts, not relevant for the binary tree). This is not important in terms of dependencies, since this last set of instructions can not include any references to other variables, but it is just to ensure a correct analysis of the node.

Phase 3 - Analysis of the dependencies This phase of the algorithm is responsible for the correct identification of all the dependencies for each node of the graph. In particular, it is structured in three more subsections, each one addressing a different kind of entity that could depends on some other data. The first one, and the most complex, deals with all the nodes belonging to the graph set identified during phase 2. For each entry of the instruction list constituting the method, the analyzer looks for any node having the same bytecode position; if so, the information regarding the instruction `stack depth` of the node is used to identify the set of instructions that could provide dependencies for the considered entity. In particular, the analyzer will examine each one of such instruction and, if there are matches to a given subsets of instructions that indicates dependency from some other data (for example, load instructions), the method will find and build the entire sets of dependencies defined by the instruction itself. In order to accomplish the task, it will call the method `getDepFromBasicBlocks`, defined in the `DataDependency` class and explained better in the following section. Particular operations have to be performed if the considered node is created as result of the invocation of a method, since the dependency must also address the name of the method called (and eventually its signature). If a node is instead passed as parameter, the analyzer will provide in the results the number of parameter to which the node corresponds; moreover, if it is modified by the called method, there will be an additional node on the graph, indicating that the input node (that must be of course an object) has been someway altered within the called method (whose name will be reported in

the dependency), and thus it can be considered a new instance with respect to the previous one. When dealing with static variables, no differences are present to what previously described. The second code subsection is related to operations toward arrays and objects in general. In these cases, the result of the operation will not be translated in a new node in the graph; the dependencies that they (eventually) determine will be reflected directly on the node corresponding to the object. For example, an array could have dependencies toward other variables that are present either in the index or value expression related. Using an alternative approach, that instantiates a new node each time an operation on the array occurs, will result in an exploding growth of the graph dimension, that does not carry any significative additional information with itself (for our purposes). In the result is also present an indication of the cardinality, in the case that the array is the source of the dependency; for each expression, the result node will contain the number of occurrences of each array in it (useful, for example, if considering the possibility to pre-evaluate some parts of such expression). The last case to be considered is if the invocation of a method will affect also the calling object (or possibly the calling class). If so, for each parameter used in the method call that generates dependencies the correspondent stack of instruction must be considered; then, the analysis will be executed in a very similar fashion as for normal graph nodes. The information about which parameters (eventually) modify the calling object is available as result of the invoked method analysis performed during phase 2.

Considering the dependencies from conditional statements One of the outputs generated in phase 1 was the complete list of conditional branches, each one with the bytecode position and the range of validity in the code; hence, for each bytecode position, there could be from zero to n valid conditions. Each node having exactly that position will thus have dependencies from each one of the valid conditions (and hence the correspondent `condition` nodes will have outgoing arcs towards such node). The function responsible for the task (`updateBranchDep`) is implemented in the `DataDependency class`.

Obtaining the dependencies generated by an instruction The task is performed with the method `getDepFromBasicBlock`, implemented in the already cited `DataDependency` class. It requires as inputs, among some data structures (such as the dependencies graph, the control flow graph, etc), the memory location, the bytecode position and the block number in which resides the instruction that generates the dependency (the source node on

the graph); for the target of the dependency, the only requirement is the key that univocally identifies the node in the dependencies graph. Basically, the algorithm performs a recursive search on the control flow graph, in order to identify the previous variables that have the same memory location on the stack as the input. In order to do this, the blocks that constitute the flow graph are examined backwards (starting from the one in which resides the original instruction), until all possible inputs have been examined. To prevent infinite loops, a list of visited blocks has to be update through the process. If no reference is found up to the first block of the method, the desired variable is likely to be a parameter of the invoked method, and hence the list of inputs must be considered. If, after this step, still no reference has been encountered, the only motivation is that the Java class has not been correctly compiled. When a match is found, the target node's dependencies list will be updated (this information will be used later to build the set of arcs of the dependencies graph).To speed up the operations, an additional method in the same class has been provided, under the name of `getLastVarFromBlockMemLoc`; as the name suggests, for a given block number, bytecode position and memory location, it provides the last variable meeting the requirements in the specified block. This can be used in order to prevent an additional, full scan on the instruction list of the block. When dealing with static variables, a slightly different approach is required, since the memory location information is meaningless for them; instead, to identify univocally such variables, the information about the name should be used. As already explained, it could be accessed using BCEL, and in particular the `getFieldName` method on instructions dealing with static elements. Two methods have been implemented in the `DataDependency` class to perform the search for dependencies; they work in a very similar fashion as the ones described in the previous paragraph, and hence will not be further examined. The only additional information regards the name of the nodes added to the dependencies list; for non-static variables, the node name (that is equal to the node key) is enough, since it is univocal in the graph; for static variables this is not true (as described in phase 2) and hence the key must always be considered before the name.

Phase 4 - Finalization In this phase, the last operations required by the algorithm are performed, and all the results are wrapped up and returned by the application. First of all, the graph is completed by instantiating all its arcs and then connecting them to the right nodes. This tasks are performed respectively by the `buildDarcs` and `completeGraph` functions, both

implemented in the `DataDependency` class; the first one instantiates the arcs using the information residing in the dependencies list of each node (which have been allocated in phase 3), while the second one connects both kind of entities together. The result is a full specified graph, where each node has two sets of arcs, inputs and outputs. Moreover, three additional lists of information are developed for the method; they show the dependencies from the parameters respectively toward the parameters of the method themselves, the object which calls it, and the return value. In order to do this, the `getParamsDep` instruction examines the graph, starting from the point of interest (i.e., the return values) and navigates the graph backwards towards the parameters; if one of them is reached in the process, it will be added to the correspondent list. This can be useful, for example, in the analysis of pure function, in which not each one of the parameters actually is involved in the definition of the result; it can be hence discarded, resulting in a decrease of the dimensional space of the input.
Having the graph completed and fully descripted, this last phase provide both a textual and graphical representation of it. In particular, the `printDepGraph` function prints on console the list of nodes belonging to the graph, each one with all the corresponding information and the list of dependencies. Moreover, a more intuitive and practical representation is available using the visualization tool Graphviz. The function `buildDepGraph` take care of converting all the data results in the right format required by files `.gv`; graphical preferences (for instance, colors and shapes) can also be modified in a second time, using a common text editor and the syntax provided by Graphviz. Once the file is completed, it can be compiled by the tool, which automatically builds and exports the image representing the dependencies graph. The last operation take care of building the `MethodSummary` object, in which all the results of the analysis are stored, that will be returned by the method.

5.2.2 Purity Analysis Algorithm

The `PureFunctionAnalyzer` class is responsible for the analysis of a method in terms of purity. It considers each kind of possible impurities that prevent the method to be considered pure, making it the target of possible bytecode resolutions. The final analyzer relaxes some constraints necessary to our previous work [6], in order to expand the set of pure functions, without impacting the correctness of the process. All the information resulting from this process are stored in a particular data structure called `PuritySummary`, that will be associated to the `MethodSummary` cor-

Data: Method name, Class name, Method signature
Result: Purity information for the method

Initialization;
for *all the possible impurities* **do**
  Perform the relative check;
end
Determine the purity of the method;
Finalization;

Algorithm 2: Purity analysis algorithm.

responding to the method under exam.

Implemented Algorithm

The following paragraphs describe in detail each type of impurities, and the techniques used to analyze a generic method; the obtained information are used to determine the characteristics, in terms of purity, of the function under exam. Such results are summarized in a `PuritySummary` object, which will be related to the correspondent `MethodSummary`, in order to group all the data available for the method. The full list of sources of impurities that have been identified includes issues regarding:

- the method signature
- the return type
- access to static variables
- access to global variables
- modification of object parameters
- invocation of non pure methods
- other sources of impurity

Preliminary operations As first task, the analyzer retrieves the resources of interest through the BCEL APIs; these include the properties of the method under exam, as well as its instruction list. In the event that the latter can not be retrieved for any reason, the method is assumed as non pure; on the other hand, if the method is declared as `native` (meaning that it is implemented in an independent way from the platform and accessible through the `Java Native Interface`), it is considered pure.

Signature check (flag `sign`) The traditional framework considers a method pure if and only if all its parameters are of primitive types; this excludes any kind of object or data passed for address, including for example array of primitive types. The current implementation, instead, admits the presence of mono or multidimensional array of primitive types (even if they could determine an explosion in terms of space of the parameters domain), while generic objects are still not suitable for this purpose.

Return type check (flag `ret`) The assumptions are exactly the same as for the signature. Primitive types, array of primitive types and the `VOID` type are allowed in terms of purity. Generic types determine the method as non pure, but should be noted that the calling function could still be memoizable (see Section 4.1.1 for additional details).

Static variables check (flags `s_r`, `s_w`) The access in read and/or write to static variables prevents the method to be pure. In fact, these values can not be retrieved from the memoization process, and hence the result obtained is not reliable. To determine these conditions, the analyzer scans the entire instruction list for the targeted bytecode instructions; in particular, the reading of a static variable is determined by a `GETSTATIC` instruction, while the writing is identified by a `PUTSTATIC` statement. The analyzer, moreover, can provide a count of the variables involved, to give a measure of the problem. The result of the operation consists in two flags, one related to reading and one related to writing towards static variables.

Global variables check (flags `f_r`, `f_w`, `f_c`) Similarly to static variables, the access to fields of objects, defined outside the scope of the method, is a problem for the purity of a function. The additional complexity, in this case, resides in the fact that such variables are not absolute (as static ones), but are related to specific instances of objects. Hence, the analyzer has to look for the correspondent bytecode instructions (`GETFIELD` and `PUTFIELD`), plus the local address of the associated `ALOAD` statement. This is the only way to determine if an access has been made towards a parameter, or an object created within the method; obviously, this second case does not constitute a problem in terms of purity.
To accomplish the task, the analyzer has to rebuild the expression of such instructions; `GETFIELDs` immediately require the `ALOAD` instruction for the object considered, while `PUTFIELDs` need also the expression of the value between them and the `ALOAD` of reference. The result of the operation is composed of three flags: one identifying the presence of reading

towards global variables, one related to writing, and one related to both operations, but specifically performed only for the calling object (situation possible only for non static methods, in which the address associated to the `ALOAD` instruction is 0); this last discrimination is useful, since the resolution of this case is less expensive, compared to the modifications required for writing access of fields belonging to objects passed as parameters.

Parameter objects check (flag `obj`) The assumptions described in the signature check process make possible the presence of multidimensional arrays of primitive types as input. However, if their state changes following the execution of the method, the memoization approach will prevent the correctness of the result. Hence, the analyzer will check that each access to the mentioned parameters will be in reading only. In the presence of writing operations on them, the method can not be considered pure. More in detail, the analyzer examines: each `ASTORE` instruction, to verify that the memory location correspondent to the parameter is not used for other variables; each `cell store`, to verify that the considered array does not have any of its cells changed by value. To ensure this last property, the regular expressions required by the instruction should be considered. Such opcode types require: the expression of the value to be saved, the expression of the value of the index of the cell, and finally the expression of the array interested by the change (the point of interest for this analysis). Operations towards generic objects are not verified, since the impurity is already been granted from the signature check.

Invoked methods check (flag `invk`) According to the traditional definition of purity, a method can not be considered pure if at least one of the called method is not pure. The analyzer checks each calling instruction, and looks for the relative information in the local repository and (eventually) in the database. If the method is not pure, its data is added to a local list of non pure methods; clearly, the calling method will be pure under this point of view if this list results empty at the end of the process. Moreover, the analyzer checks again the list in order to identify which subset of such methods are however acceptable (for the difference between pure and memoizable, see Section 4.1.1). If all the non pure methods are acceptable, the calling method could be memoizable; on the other hand, if at least one of them does not satisfy the requirement, the calling method can not be correctly memoized. A called method is accepted if its only source of impurity is related to the return type; however, it should be also verified that the object returned by it is not further returned by the calling method. Failure in any

one of these checks prevents the call to be acceptable.

Other impurities check (flag `other`) This check deals with the presence of the last prohibited instruction of the traditional definition,the opcode `INVOKEINTERFACE`. The presence of the instruction could be a problem in terms of purity if it refers to an object passed as parameter to the calling method. However, since the check is already performed with regards to the signature, this information could be considered redundant. Other impurities that could be eventually identified and that can not be actually solved should be grouped under this section.

5.3 Bytecode Modification Module

Given the analysis results, the module execute the necessary instructions that permits to inject the code needed by the `Decision maker` meta-module. A function is identified by a `methodId` string, which is composed by the `className` in which the method resides (comprehensive of the full class path), the `methodName` itself, and its signature. The insertion of the `Decision Maker` module is performed at bytecode instruction level, and hence does not require an additional compilation of the code. The definition of such component as meta-module refers to the fact that it is, in reality, the full set of instructions that ensure the correct execution and integration of the proposed approach. Effectively, the `Bytecode modification` module injects in the original method two blocks of code: one related to the `Lookup` on the relative table (in which the results of the function are stored), and another one related to the `Trade off` element. The result of the operation consists in a new version of the method, which is now composed of three main blocks:

- `Lookup` block in which the result table is searched for a value in correspondence of the provided inputs;
- original block of instructions of the method;
- `Trade off` block, which manages the results table through the API provided by the relative module.

As depicted in Figure 5.2, in the first step Java bytecode methods are statically analyzed to detect whether they are pure functions or not. Code is analyzed by means of BCEL [21], a Java library for bytecode inspection, in order to identify bytecode instructions classified as impure, i.e. which make the method impure.

As illustrated in the previous section, the proposed application need to operate at bytecode level, to maximize the applicability of such operations. It is hence necessary to understand the principles of the bytecode modification, how it can be effectively applied, and the techniques that permits to achieve the objectives of interest in terms of re-engineering of pure functions.

5.3.1 General Concepts

The previous section has already introduced the BCEL library and how it can be useful for bytecode analysis. In this section it is used to modify the list of instructions that compose a method in a compiled Java class. In particular, BCEL provides a set of APIs to manage the `InstructionHandles` that form the instruction list, and that permits to create new instructions and move or delete existing ones. For the first task, the `InstructionFactory` class exports various methods to create new bytecode instructions, as well as a pool of instructions "off the shelf", ready to be injected; for the second, the `InstructionList` class provides a wide variety of functions to manage every single instruction handle, as well as the entire list.

The following list reports some of the main aspects that have to be kept in mind when approaching bytecode modification:

- there is not any kind of verifier at compile time when modifying the instruction list of a method; hence, a syntactically wrong change usually results in a class that can not be executed, since it can not be compiled. A good practice is to first verify the new dumped class, and then check it for the expected behavior;
- after modifying a method and obtaining its correspondent instruction list, a good strategy is to remove the old method and add the new one to the class, instead of simply replacing it. This could in fact lead to problems, especially regarding the local variable table, that prevent the method to be executed;
- after defining the new method with all the resources it needs (instruction list, signature, visibility, etc), and before adding it to existent class, some clean up methods have to be executed in order to guarantee the correct execution of the new function. In particular, for each new method, these three operations have to be performed:
  - `setMaxStack()`: computes maximum stack size by performing control flow analysis;

- `setMaxLocals()`:computes maximum number of local variables;
- `removeLineNumbers()`:removes all line numbers.

It should also be noted that an additional, important assumption has been made, prior the presented work; to prevent problems regarding the uncontrolled access to resources that could be modified, such as global variables, the execution is always considered as single-threaded. The modifications illustrated in the next sections can work as well as when considering a resource lock at method granularity; basically, the lock on a resource is taken at the beginning of a function, and then released only after its termination. In this way, no external process can actually read or write to the variables involved, since it is not possible for them to acquire such lock. In a situation in where, instead, the lock is granted with smaller granularity (i.e., single operation), uncontrolled behaviors could arise, since the modifications presented grant a correct status of the considered variables only after the completion of the method, and not during its execution.

5.3.2 Resolution of Impurities

In the following section, we describe how we act in order to automatically modify the application bytecode to resolve the impurities founded during the bytecode analysis. The impurities that can be resolved are: the use of static variables, the use of field variables, and the invocation of non pure methods. For each of these impurities are defined the steps in order to create (or modify) a sub-function that can be considered as a pure function.

Static Variables

Primitive types of static variables are good examples of how bytecode modification can be used to deal with non pure methods. The objective of the work is to wrap a subsection of the instructions that compose the original method in a new, pure function (which can be eventually memoized), and call it from the original method, in order to guarantee the original behavior. In fact, at the end of the operations, no difference has to be spotted at run time from the two versions, in terms of results and produced effects. Having the graph that describes the data dependencies of a method (as a result of the analysis presented in Section 5.2.1), it is quite easy to determine the presence of static variables that define the method as non pure; in fact, they are translated in the graph as nodes having the name starting with *s*, and are eventually underlined with a different color. The presence

of outgoing arcs from such nodes means that their value is been used from some computation, while incoming arcs identify changes in their states. As already reported, at the end of the execution not only the result, but all the static variables in play, must have a consistent value with respect toward the original method.

The final result of all the operations described later consists in a new method, having exactly the same signature as the original one, that is by nature non pure (since it reads/write towards static variables). During its execution, it calls a new method that consists of the computational part of the original method; it is however been modified in such a way that it will result pure, since all the non purities are now managed by the wrapper (before and/or after the call to the pure method). In particular, the original method will now have the following structure: the invocation of the pure method, with the correspondent stack of parameters (the original ones, plus the static variables loaded directly on the stack); the updates of the static variables that have been modified during the execution (retrieved from the array variable returned by the pure method); the return of the original value produced by the method (also retrieved from the mentioned array). The structure of the pure method will be the following: the creation of the array structure, that will be returned by the method; all the original computation that compose the original method; the saving of the return value and the final values of the static variables modified in such array; the return of the already mentioned array. The location of the variables within it will be fixed for a better management of the method stack.

The following code snippets summarize the structure of the methods resulting from the application of the presented solver.

```
originalFunction (original pars){
  //Acquisition of required static variables
  result_set = pureFunction(orig_pars, static_pars);

  //Update of modified static variables
  return final_result;
}
```

```
pureFunction(orig_params, static_pars){
  //Original computation (using local variables)
  //Results saving
  return;
}
```

The following paragraphs describe in detail the steps necessary to solve the impurities presented. Each step describes the elementary operations needed to achieve the final result, with both theoretical and technical (in terms of BCEL instructions) references.

Static variables mapping After all the resources needed with the BCEL library, such as the class and method to be modified, have been located, the first operation is to build a mapping between the static variables considered and local memory locations that can be used for operations upon them. After having obtained the first available memory index (which can be determined examining all the instruction list once), a new location is assigned to each static variable (univocally identified by its name), remembering that the type `double` actually requires two of that. The assignment is realized with a first come, first served behavior; however, to facilitate another operation examined later, all the static variables whose value is read before any (eventual) modification are assigned first. The solver keeps a local repository of `StaticReference` that describes each one of the mapped entries; it can be used in any time to retrieve all the information needed regarding a static variable involved in the process.

Bytecode substitution Now it is possible to manage all the `PUTSTATIC` and `GETSTATIC` instructions; in particular, each `PUTSTATIC` will be substituted by a correspondent `store`, and each `GETSTATIC` by a correspondent `load`. This will effectively remove the causes of impurity for the original method. Moreover, during this step it is useful to save a list of all (distinct) `PUTSTATIC` instructions that are removed, since they will be needed later.

Data structure creation The next operation aims to create a structure that will save the original returned results of the method, as well as the values of all the static variables that are modified during the execution. Since the dimension of such structure is known at compile time, an array variable is the optimal choice in terms of both memory occupation and access speed; in fact, tests have demonstrated that the access time to an array is faster at least 10 times with respect, for example, an hash map structure (in which the computation of the hash required is the dominant factor in terms of time). The bytecode instructions required to instantiate a new array are: the dimension (equivalent to the total of static variables that will need update, plus the original return value of the method), the `NEWARRAY` instruction, and an `ASTORE` with the first available index of memory (that will also be saved in the mapping defined in the first step, with an key equal to `ResultsArray`). These three instructions will be injected right at the beginning of the instruction list.

Final values saving Having the support structure to save the final values of the static variables modified in the execution, it is necessary to insert the instructions that will effectively store the last value of each one of them in the array of results. In order to do that, the following list of bytecode instructions are needed: `ALOAD` for the array (index available in the mapping map, under `ResultsArray`), index of the element (the order is the same as the `PUTSTATIC` instructions previously saved), the `load` that retrieves the value from the local variable mapped to the static one (index available in the mapping), an eventual cast to manage conversions between types, and a `array store` statement (i.e., `IASTORE`); for the complete list of such operators refer to the tables in appendix A. Regarding the original value returned by the method, as first operation a `store` instructions must be inserted before the final return, in order to have a local copy of the variable (instead of immediately returning it); the index for this `store` is the first available, and will be saved in the mapping as `OriginalRetValue`. Then, it's necessary to perform the same operations described for static variables, in order to save that value in the array of results; in particular, for convention it will be stored in position 0 of such array. This step of the algorithm must be repeated for each return statement of the method, since the information have to be available and correctly saved regardless of the execution flow of the function.

Data structure saving The array of saved values will be the object returned by the new, pure method. To save it, it is necessary to introduce an additional `ASTORE` before the last return statement of the method; as index, it is possible to use the same as before (`ResultsArray` in the mapping).

Static variables update After that, it is possible to insert all the instructions needed to effectively update the values of the static variables modified. The bytecode instructions needed for each variable, in order to accomplish that, are: `ALOAD` of the array of results, index of the element in the array, eventual cast between types, `array load` statement, and the original `PUTSTATIC` correspondent to the index (previously saved). The process follows the conventions previously described. Moreover, it is necessary to load on the stack the final return value of the method, which will be passed to the original return. The operations needed are exactly the same as for static variables, whithout, the `PUTSTATIC` instruction (remember that the value of interest is stored in position 0 of the array).

Pure method creation It is now necessary to effectively create the new, pure method, and to adjust the instruction list of the original one (which will be at the end a sort of wrapper for the pure one). First of all, the signature for the new method must be defined. The parameters that it will require are the original ones, plus all the the static variables that are read only (or, in any case, read before they are be modified); the array of correspondent types can be obtained combining the original parameters and the static variables previously mapped. The return type will be generally a generic object, but it could be more specific when dealing, for example, only with integer values (in that case, the method will return an array of integers). The name of the new method will be the original one plus the suffix `Pure`, while the class will be clearly the same. The last operations regards the instruction list for both the original and the new method. First of all, a copy of the current one has to be made; different sets of instruction handles must be cut out from each one of them, in order to realize the two, distinct methods. In particular, for the pure one the last part of the instruction list must be removed; more in detail, the section to be removed will begin with the `ASTORE` for such array (index retrievable from the mapping under `ResultsArray`). The wrapper method will behave in a specular manner, so the first part of the instruction list (with the same limit defined) must be removed. The last task is the effective creation of the pure method; using the `MethodGen` class, and all the resources defined before (including the new instruction list), it is possible to create the new, pure method instance. The new method will have the original access flags (private, protected, public) and the additional property of being static, if not already present. Note that at this point of the algorithm the actual version of the new method is not complete and can not even be compiled.

Return statements substitution As already explained, the new method will return an array, and hence a variable of type object. It is necessary to introduce, at the end of its instruction list, an `ARETURN` instruction. Moreover, all the original returns eventually present in the method have to be changed in `ARETURN` (if not already so); otherwise, the return type of the method will be invalid and hence it can not be compiled.

Memory locations fixing Another operation to be performed on the new method regards the fixing of memory location. Assume for example that the original method has two parameters, plus the implicit caller. The new method will require also an additional static variable (mapped for example on location 7) as parameter. Since this will be in fact the fourth parameter, in the

pure method it will have assigned the memory location 3 (assuming that the original parameters are not of `double` type); the correspondent load for such variable, defined in the mapping, has location 7. To address the discrepancy, the following operations are performed. First, if the original method is not static, all the local addresses must be translated backwards by 1 (since the address of the implicit caller, always in position 0, is not valid anymore); after that, the application calculates the last address used by the original parameters, and hence the first available for the new ones (corresponding to the static variables). Moreover, the solver calculates the exact amount of addresses needed by such parameters, and save the information under the variable `static_req`. Having the result, all the local variables having local address higher than the last used by the original parameters, are translated forward exactly by `static_req` positions. The last task is to change the addresses of the `load` and `store` instruction inserted during the substitution step, in order to be consistent with the new locations of such variables. At the end, the mapping is updated to reflect the changes operated during this phase. This method of addresses management has the advantage of a better use of such locations, since all the values between 0 and the last one are effectively used (there is not any non-used location is such range).

Branch instructions fixing The final operation that has to be performed for the pure method is the fixing related to branch instructions. Ideally, since the substitution of instructions has been 1 to 1, this is not required, but it is good practice (and mandatory for the solvers presented later). In this case there are no discrepancies between the number of instructions handled at the start and at the end of the modification process, as far as the branch instructions are involved. However, it is suggested to set again the target of such kind of instructions in order to avoid potential conflicts. The operation can be performed manually, or through the use of the `redirectBranch` method provided by BCEL. Clearly, before any changes of the method, it is necessary to save in a separate data structure the complete list of original targets for the involved branch instructions.

Pure method invocation Two additional operations have to be executed with regards to the original method (the non pure one): building the parameters sequence that will compose the stack for the pure method call, and inserting the call itself; these bytecode injections will be located exactly at the beginning of the instruction list for the wrapper method. To build the stack, all the `load` instructions correspondent to the parameters of the

pure method must be instantiated. Since the invoked method will be pure by construction, and hence has been specified as static, the load correspondent to the (eventual) caller it is not required, so the first operation is to load all the original parameters, in the original order, on the stack; then, the static variables must be loaded (the original `GETSTATIC` instructions, in the right order, are required). In the event that the static variables considered are not public, the proposed solution will generate an `Illegal Access Exception`, hence an alternative approach is required. Basically, it is necessary to implement the getters for the static variables needed, in the class in which they belong. These methods will be static and public, and return the value of the variables needed; hence, instead of putting directly the `GETSTATIC` instructions, it is necessary to call such functions directly on the stack, providing the opcode instructions that generate the invocations needed. The stack is now complete; the last operation is to create the call for the pure method, hence the `INVOKESTATIC` correspondent bytecode instruction must be inserted. All the resources needed by the `InstructionFactory` class can be retrieved from the instance of the pure method generated (name, class, array of types representing the parameters, return type).

Final operations Both methods are now completed and the instruction list of each one is well-formed. For the wrapper method, is good practice to create a new instance of the method, having exactly the same characteristics as the original one (with the exception of the instruction list), as explained in Section 5.3.1. For both methods, the clean up methods illustrated in the same section must also be executed. As last operation, remove the old method from the class and add the new ones. The class is then ready to be dumped directly with a `.class` extension.

The presented solver could be extended in order to manage also array (mono or multidimensional) of static variables. The concept used will be very similar to the illustrated algorithm; however, the data structure needed to save the values of such variables should be substituted by a more complex one, since it is now necessary to save not only the location of the variable, but also the index of the location within it. For example, it could be used an array having *n+1* dimensions with respect to the static one, in order to save the value, the location and all the indexes needed, or a simple array in which each element is a tuple (properly formed) that contains all the required information.

The presence of static objects, on the other hand, it is more difficult to approach. The proposed algorithm, in fact, will operate on a copy of such object; however, since it will be passed by address, the pure method will operate directly on it, thus resulting non-pure. To effectively deal with such situations, it is necessary to understand which operations are made upon the object (or that influence the object itself), in order to correctly manage them.

Field Variables

As explained in Section 5.2.2, the access to public fields of an object is a factor that determines the non purity of a method; in fact, such variables are by nature defined externally respect to the considered method, and hence the method could be non deterministic (towards the parameters it takes) and/or produces side effects (i.e., updates of such fields). Given the (conservative) assumption that a pure method can not take objects as parameters, a subset of the problem of transform a non pure function in a pure one can be individuated. The following section considers how the impurities of a method, related to the access to the callers public fields (of primitive types), can be solved in order to obtain a a subsection of the method which respects the requirements of purity, and could be hence memoized. For the resolution of the scenario in which are involved fields of objects passed as parameters to the selected method, see the next section.

The final result of all the operations described consists in a new method, having exactly the same signature as the original one, that is by nature non pure (since it reads/write using static variables). During its execution, it calls a new method, which consists in the computational part of the original method; it is however been modified in such a way that it will result pure, since all the non purities are now managed by the wrapper (before and/or after the call to the pure method). The structure of the result generated by the algorithm is almost analogue to the one obtained for static variables, since the methodology is very similar for the two cases. The original method will have the following structure:

- the invocation of the pure method, with the correspondent stack of parameters (the original ones, plus the field variables needed);
- the updates of the field variables that have been modified during the execution (retrieved from the array variable returned by the pure method);
- the return of the original value produced by the method (also retrieved from the mentioned array).

The structure of the pure method will be the following:

- the creation of the array structure, that will be returned by the method;
- all the original computation that composes the original method; the saving of the return value and the final values of the global variables modified in the created array;
- the return of the already mentioned array.

The two solvers presented could be combined in order to manage situations in which there are accesses to the fields of static objects. The impurities are still managed by the wrapper method (on the stack before the call to the pure method, and after its execution); the difference, during the substitution of instructions, will be related to the fact that instead of the `ALOAD_0` instruction, the target of the `GETFIELD` statement will be a `GETSTATIC` that refers to the static object.

The following code snippets summarize the structure of the methods resulting from the application of the presented solver.

```
originalFunction (original pars){
  //Acquisition of required field variables
  result_set = pureFunction(orig_pars, field_pars);

  //Update of modified field variables
  return final_result;
}
```

```
pureFunction(orig_params, field_pars){
  //Original computation (using local variables)
  //Results saving
  return;
}
```

The implemented algorithm that achieves the result is very similar to the one related to the management of static variables, with small changes to address the differences between the two entities. The following description reports all the steps required, in terms of operations; for additional information refer, when it is possible, to the detailed description available in Section 5.1.

Field variables mapping The first operation is basically the same of the algorithm that manages static variables; this time the mapping will be instead between fields of the calling object and memory locations. Keep in mind that the access to a public field (bytecode instructions `GETFIELD` and `PUTFIELD`) requires as additional information the memory location of the

object which the instruction refers (`ALOAD` statement); in this case, the algorithm looks for the correspondent `ALOAD_0` instruction, which identifies the calling object. `PUTFIELD` and `GETFIELD` instructions that refers to other object are ignored by the algorithm, since (due the assumption that no object is passed as parameter to the method) they will refer to variables defined internally, and hence that do not influence the purity of the method. The solver keeps a local repository of `FieldReference` that describes each one of the mapped entries; it can be used in any time to retrieve all the information needed regarding a field variable involved in the process.

Bytecode substitution it is now possible to apply the substitutions that will eliminate the non purities caused by the access to field variables. More in detail, for each `GETFIELD` that refers to the calling object, a correspondent `load` instruction must be inserted; for each `PUTFIELD` that refers to the calling object, a correspondent `store` must be inserted. Keep in mind that the referred `ALOAD_0` instruction must also be deleted, otherwise the instruction list will not result well formed, and hence the class will not be compiled nor executed correctly. In place of such instructions, the solver inserts `NOP` statements, that will be useful later on when dealing with the update of branch instructions.

Data structure creation The next step is the allocation of the array in which all the last values of the modified fields and the original return value of the method will be saved. The instructions to be inserted are exactly the same as for static variables.

Final values saving After having initialized the support structure for all the values of interest, it is possible to save them. The process is straightforward the same as for static variables; for convention, the original returned value of the method is still saved in position 0 of such array, while the other field variables follows the order used in the mapping defined during the first step of the algorithm.

Data structure saving This operation insert the `ASTORE` instruction to save the array previously defined.

Field variables update The following operation takes care of inserting the instructions required to update the value of fields that have been modified, as well as load on the stack the result of the method (which will be returned by the final return of the wrapper). The process is basically the same as

for static variables, using PUTFIELD instructions instead of PUTSTATIC statements; in this case an additional ALOAD_0 has to be inserted as reference for the PUTFIELD instruction. The final result to successfully update the value of a field will be: ALOAD_0 for the PUTFIELD instruction, ALOAD of the array of results, index of the element in the array, eventual cast between types, array load statement, and the original PUTFIELD correspondent to the index (previously saved). The conventions are the same as previously explained.

Pure method creation it is now necessary to effectively create the pure method (which could be the target of the memoization process). It will have as inputs all the original parameters, plus the field variables mapped, and return type as object (since it will return the array of saved values). It will include all the computational part of the original method, that has been modified in such a way that now results pure. Then, the instruction list for both the pure method and the wrapper are defined, in the same way as for static variables. The operations required to achieve the goal are exactly the same as described in Section 5.1.

Return statements substitution As already explained, the new method will return an array, and hence a variable of type object. It is necessary to introduce, at the end of its instruction list, an ARETURN instruction. Moreover, all the original returns eventually present in the method have to be changed in ARETURN (if not already so); otherwise, the return type of the method will be invalid and hence it can not be compiled.

Memory locations fixing To finalize the instruction list of the new, pure method, a fixing of the memory location of some instruction handles must be performed. The problem arises since the JVM automatically defines an index of memory for the new parameters, that is not consistent with the mapping previously defined. The problem (and its resolution) is described more in detail in Section 5.1, since it is exactly the same situation that happens when dealing with static variables. The instruction list for the pure method is now complete and well formed.

Branch instructions fixing An additional operation that has to be performed for the pure method is the fixing related to branch instructions. This situation differs with the one presented for static variables, since the substitution of instructions has not been 1 to 1 (a load/store instead of the correspondent GETFIELD/PUTFIELD); however, since an additional NOP was

inserted to compensate the total number of instruction handles, it is possible to act exactly as for static variables. The only additional operation consists in the call of the `removeNOP` method (provided by BCEL), which deletes all such opcodes and takes care of managing the branch instructions influenced by the process.

Pure method invocation Regarding the wrapping method, the last operation to be performed on its instruction list is the invocation of the pure method. In order to accomplish this, the stack of the parameters required by the new method must be build, and the instruction responsible for the invocation of such function must be inserted right after it. The stack is realized by loading on the stack: the list of original parameters, in the same order; the list of fields needed by the pure method, in the same order. Similarly as for static variables, if a field is public, the pair `ALOAD-GETFIELD` is enough; if not, it is necessary to implement an additional getter in the class of which the object belong, and then call the function directly on the stack (providing the `ALOAD` that refers to the caller, since such method can not be static). The stack is now ready, and hence the bytecode invoke instruction must be inserted.

Final operations Both methods are now complete and the instruction list of each one is well-formed. For the wrapper method, is good practice to create a new instance of the method, having exactly the same characteristics as the original one (with the exception of the instruction list). For both methods, the clean up methods illustrated in the same section must also be executed. As last operation, remove the old method from the class and add the new ones. The class is then ready to be dumped directly with a `.class extension`.

The previous solver could be adapted to deal with the access of fields belonging to objects passed as parameters. The additional complexity, in this case, resides in the fact that it is necessary to understand when a `GETFIELD` or `PUTFIELD` is related to an `ALOAD` instruction that refers to an object given as input, and not instantiated by the method itself. The `FieldReference` structure that univocally identifies a field takes in account also the location of the object which it belongs (that has to be computed prior any modification). If the saving of the final values of the fields involved follows the order of declaration, the presented array structure is enough to store and retrieve the values of the fields to be updated; it is just important to remember to relate the correct `ALOAD` instruction for each

PUTFIELD needed.

An interesting scenario is the one in which either static variables or fields are accessed only in reading (that is, their value at the end of the method is exactly the one they had at the beginning). This situation can be solved with less overhead, and permits to fulfill an important property. The algorithm that solves the problem is a subset of the ones already presented; however, there is no need to save and update any value, hence the return type of the pure method can be exactly the original one (since it will return just the original return value). This results in the external method being exactly a wrapper, since no other operations is executed, with the exception of the call to the pure method. As presented in the following section, it is possible, for a method that calls such wrapper (that clearly results non pure) to call directly the pure method within it, translating to an higher level the source of impurities. With the assumption of single threaded execution, the methodology can be applied even when it is necessary to update global variables, at the cost of an increased complexity in terms of resolution.

Invocation of Non-Pure Methods

The last case to consider, with respect to the operating definition of pure function (see Section 5.2.2), deals with the presence of functions called within a method. By definition, a method can not be considered pure if at least one of the methods invoked during its execution is generically non-pure. The implemented solver for such issue examines the instruction list of a method and analyzes every instruction that generate a call to another method (in particular, the INVOKEVIRTUAL, INVOKESPECIAL and INVOKESTATIC statements). For each one of them, the framework searches the external database, looking for information about the purity of such methods; in the eventuality that no information is available for the considered function, the assumption is that the method is non pure until further analysis. Three different situations could be verified in the process:

- the method is not pure; no additional operations can be performed, since the method results non memoizable. The method that calls such function will be considered non pure, unless further techniques are applied (i.e., partial memoization);
- the method is pure; no additional operations need to be performed;
- the method is not pure, but it is only a wrapper for a pure method; the issue can be solved with the algorithm presented in the following

section. The idea is to substitute the call to the wrapper with the direct call to the pure method that it has within it; in fact, by definition a method can be considered a `wrapper` if it calls a pure method within it, and does not execute additional operations.

The final result of the algorithm described below will be the original method, having (when possible) the calls to non pure methods substituted by calls to pure ones (with the relative stacks). No other modification is introduced from the the solver.

The main difficult with this kind of resolution is the construction of the stack for the substituted methods. In order to successfully inject the instructions required for the loading of the required resources, a correspondence between the remote memory locations interested and the local ones (with respect to the analyzed method) is necessary. The following algorithm shows how such resolution is possible; the actual version is referred to a pure function that takes fields as additional inputs. Similarly, and in an easier way, it can be applied also for static variables (the decreased difficulty resides in the fact that the location for such variables is global, and hence the instructions needed to load them, are exactly the same irrespective the context considered).

Resources initialization Similarly to the algorithms presented in this chapter, the first operation is to retrieve all the resources needed to modify the bytecode of the selected method. This set includes also the list of targets for each branch instruction present in the instruction list; this variable will be used when fixing the related instruction handles, following insertions/deletions in such list.

New stack creation The first operation is to build the stack needed by the pure method that will be called. For each parameter that will be located on the stack, the correspondent instruction list is created; hence, the final result will consist in a full set of such instruction lists, in the correct order. In the event that the original called method has generic objects in its parameter list (typically, they are required as reference for the `GETFIELD` instructions), the solver removes them from the parameter list and recompiles the address locations for them. Then, it adds on the stack the fields needed; to do that, it creates the correspondent `GETFIELD` instruction, or the call to the getter that provides the variable (if the field is not public, as illustrated in Section 5.3.2). The references for such instructions must be computed, since usually the calling method has assigned a different location for the

objects needed. A similar approach is used for the required static variables; however, this operation has reduced complexity since it is not necessary to deal with the memory locations of the referred objects. At the end of the process, the stack will consists in a sequence of instruction handles lists, each one providing one parameter for the new invocation instruction.

Stack substitution The next step its the substitution of the original stack, with the instruction list created as result of the previous phase. The dimension of the original stack is hence calculated, in order to remove it from the method's instruction list; the new sequence of instruction handles is injected in its place. The difference in number of statements between the two stacks is necessary for the next phase, and hence must be saved.

Branch instructions fixing Similarly to the solvers presented in this chapter, it is necessary to deal with the branch instructions that compose the method. However, this case is more complex, since the numerical difference between the original instruction list and the modified one is not known beforehand. Hence, every time a stack substitution is verified, the target of the instruction branches must be translated forward (or backward) in relation to the difference, in terms of number of instruction handles, between the two versions (the variable saved at the end of the previous step). Every branch instruction in the list is considered: if the target of such instruction is preceding the injection point, it is sufficient to set again the same target; if, on the other hand, the target is subsequent the point of interest, it must be set again according to the shift value previously saved. This operation is possible implementing the correspondent code, or using the `redirectBranches` method, provided by BCEL.

Invocation of the new method The last operation to be performed is the insertion of the new invocation instruction. The old one is removed, while the new one is first created, and then inserted exactly in the same place of the old one (right after the stack previously instantiated). After that, the solver starts to check again the instruction list and, if possible, iterate the algorithm for the other calls that satisfy the requirements.

Extending the approach - Inlining

The described approach can be extended to reach even broader applicability. The current version allows the substitution of a call to a method, if the correspondent pure version exists (and hence, the called method can be

considered as a `wrapper`). However, such approach could be applied in a similar fashion even for methods that are not `wrappers` in a strict sense. Consider for example the result produced by the solver of Section 5.3.2 for a method in which the value of some static variable has been modified. The obtained method will be composed of a pure section (implemented in a separate function) and a non pure code block, relative to the update of the static variables. Hence, it can not be considered a `wrapper` according to the definition of Section 4.1.1. A similar approach to the one proposed in the current section could be however applied. The solver needs to translate at the level of the caller not only the invocation of the pure method (providing the correspondent stack) but also the instructions required to update the interested variables. The advantage consists in the fact that the execution of the original method has now to perform one less function call, since it can refer directly to the internal instructions of such function. This methodology, which can assume the name of `inlining`, can be considered an additional step of optimization and re-engineering for a generic method. The main difficulties in such operations reside, similarly as for the solver of the current section, in the realization of the correspondence between the memory addresses of the two methods involved.

Considerations about the modification of Java APIs

The current implementation of the framework, as described in Section 5.1, does not permit the modification of methods belonging to the set of the Java API; in fact, the question could be the object of legal issues (for example, the `Java 2 Runtime Environment APIs` basically can not be overwritten, while `OpenJDK`, being under GPL license, is not affected by such considerations). However, in theory, such operations are possible, since they do not differs with respect to the ones described in the present chapter (methods and classes modification) and the previous one (jar updates). On the other hand, exposing data structures that are not meant for it, could be the source to other problems.

A couple of solutions are proposed to deal with the problem. The first one consists in a direct modification of the original class/method, in the same way as described in the chapter. The obtained version, however, will be associated to a different class path, that will be used only for the execution of the application. In terms of code writing, the modifications result hidden (since the original APIs are used), while at run time the applications will rely on the modified versions. The same strategy could be used, for example, when dealing with the resolution of impurities related to private

fields; the getters introduced by the solvers will reside in a new version of the class, used only at run time, while the original one will remain unchanged for the coding phase. The second approach, on the other hand, creates new versions for the modified classes, while preserving the old ones. Then, for a generic Java class, every call related to the old methods of such class is redirected to the one created in the modified version of it. In this way, no original APIs is effectively modified; instead, the calls to them are redirect to the custom-created methods in external classes. Given these proposals, is up to the user to decide if and how methods belonging to the Java APIs have to be modified.

5.4 Decision Maker Meta-Module

`Decision Maker` is a meta-module (i.e. a module that is abstract). The integration of *Lookup* and *Trade off* modules through the use of `Bytecode Modification`. The `Decision Maker` is the set of bytecode instructions that allow the operation and integration of all modules of the proposed approach. `Bytecode Modification` module adds `Decision Maker` meta-module, directly going and writing the necessary bytecode instructions for the interaction between the other modules without usage of an additional real module. Thus, effectively `Bytecode Modification` module also deals with the insertion in two blocks of the bytecode instruction: a block for *Lookup* on the related function table and a block for *Trade off* module call. It is interesting to note that a pure function `returnType className.methodName (params)` implemented as in Figure 5.6 (in pseudocode), will be modified after the insertion of `Decision Maker`

```
public class className {

    ReturnType methodName(Parameters p){
        ReturnType result = null;
        //ORIGINAL CODE {
            //effettua il calcolo di result
        //ORIGINAL CODE }
        return result;
    }

}
```

Figure 5.6: *`className.methodName` before inserting the `Bytecode Modification` module.*

meta-module, as shown in the pseudocode in Figure 5.7.

It is thus possible to see the code blocks highlighted in different colors:

```
public class className {

    ReturnType methodName(Parameters p){
        ReturnType result = null;

        //BLOCCO di LOOKUP
        String methodId = "className;methodName;methodSignature";
        Object[] params = {p};
        double alpha = 0.5;
        TOParameters toparam = new TOParameters(params);
        Return ret = MemoryManagement.getInstance().lookupResult(methodId, toparam);
        if (ret != null) {
            MemoryManagement.getInstance().setupFrequency(methodId, toparam);
            return (ReturnType) ret.getpValue();
        }
        long compTimeStart = System.nanoTime();

        //ORIGINAL CODE {
            //effettua il calcolo di result
        //ORIGINAL CODE }

        //BLOCCO di TRADE OFF
        long finalTime = System.nanoTime();
        long priorityTemp = (finalTime - compTimeStart);
        ret = new Return(result);
        TOFunction tof = new TOFunction(signature, toparam, ret, alpha);
        tof.setupTime(priorityTemp);
        TradeOff.tradeOff(tof);
        MemoryManagement.getInstance().setupFrequency(signature, toparam);

        return result;
    }

}
```

Figure 5.7: `className.methodName` *after the modification* `Bytecode Modification` *module.*

- Green, *Lookup* block which makes search in the function table to check if the result has already been tabulated to given input parameters of the function;
- Blue, the original function's code block;
- Yellow, *Trade off* block that deals with table management by using the API provided by *Trade off* module.

The execution flow of the function after `Decision Maker` meta-module insertion has been shown in Figure 5.8. The blocks of the flow diagram reflect the previous highlighted colors to give an idea on how the operations are distributed inside of the function.

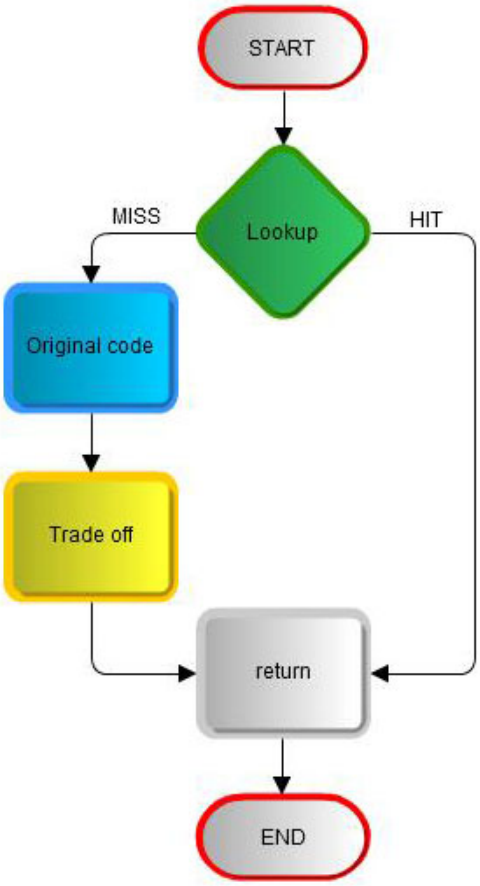


Figure 5.8: *The execution flow of a function after inserting* `Decision Maker` *meta-module.*

Lookup block Lookup block deals with the search on function table of already computed result for the current inputs of the executing function. Particularly (referring to the Listing 4 the green block), the block performs the following primary operations:

1. It generates a `methodId` string that will be used later to make a *Lookup* on the table;
2. It creates an `Object` array containing the values of the function parameters for the current implementation, and it uses this to instantiate a `TOParameters` (see Section 5.5.1) object type, which is also required for the *Lookup*;
3. It performs a *Lookup* on the table for running to see if there is a result for the method (identified by `methodId` string) performed by input parameters (saved in `toparam`);
4. If *Lookup* returns a HIT:
   (a) It updates the frequency of the function usage (increases by 1 with an internal counter) with `methodId` by calling `toparam` parameters;
   (b) The method returns the correct value taken from the table with the function and the parameters which was called from the previous *Lookup*;
5. It initializes `compTimeStart` variable for the calculation of the execution time of the function's original code.

Trade-off block The code has been internally changed juts for inserting before each return a call to *Trade off* module. The inserted code performs the following operations:

1. It calculates the execution time of the original function;
2. It creates a `Return` object with the result given by the original function calculation;
3. It creates `TOFunction` (see Section 5.5.1) object passing to signature, input parameters and result of the execution of the original function code;
4. It calls the tradeoff passing to the just created `TOFunction` object;
5. It increments the counter of the frequency to the current function call with the parameters under consideration.

5.5 Memory Management Module

5.5.1 Data Structures

During the process, the data is stored in RAM memory to access a more rapid and less expensive from the energy point of view (in fact the data saving on database or hard fisk will be inefficient and even counterproductive from time point of view and the energy consumption point of view).

For memory management, it has been used `MemoryManagement` class (see Figure 5.9). It is structured mainly with a `HashMap <String, TOFunction> mMemory`, which manages `methodId` and the related `TOFunction` pairs. In addition, it has been memorized more information

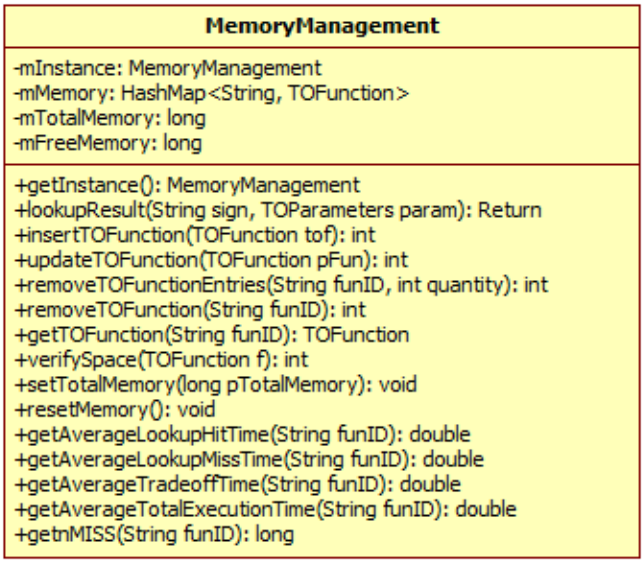


Figure 5.9: *The `MemoryManagement` class.*

about the execution times and the number of hit and miss on the tables in memory. The `MemoryManagement` class allow the use of the allocated memory to multiple pure-functions concurrently. Each pure function have its tuple inside the `mMemory` data structure. When a memoized function is called, the `Decision Maker` meta-module makes a lookup inside the `mMemory HashMap` in order to find if exist the related `TOFunction`.

In `TOFunction` (see Figure 5.10), there exist another data structure that maintain the following information:

- mValues <TOParameters, Return> - `mValues`, that manages `TOParameters` pairs (i.e., the values of passing parameters for each function call in object) and its return value is in Return type;

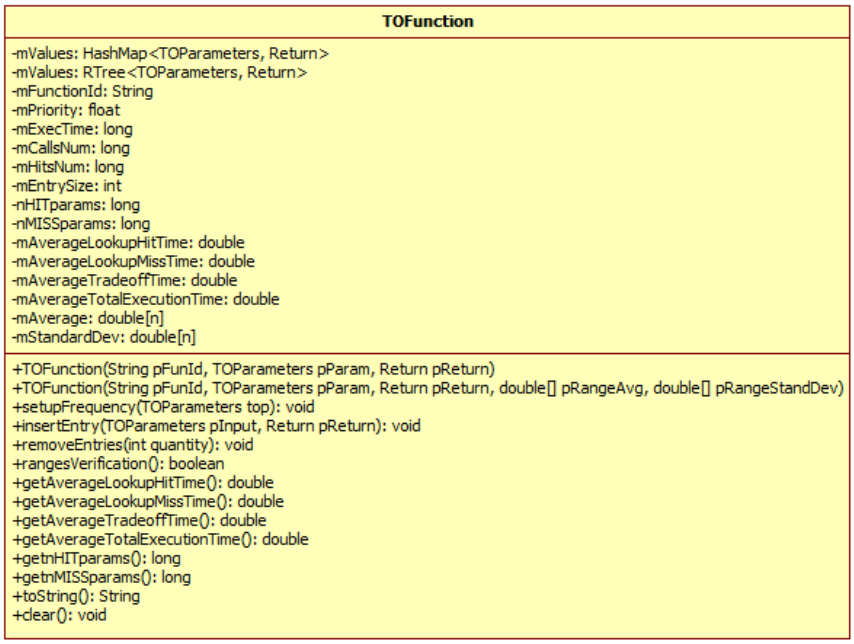


Figure 5.10: *The `TOFunction` class.*

The mValue data structure (inside the `TOFunction` class) can be an `HashMap` or a `RTree` in order to respectively implement the *precise lookup* or the *memoized SLA lookup*.

These two data structures map the `TOParameters` class (see Figure 5.11) to the return value (`Return` class). Both of the data structures allow the lookup with respect to the `TOParameters` class. The `TOParameters` class contains a set of ordered tuples that represents the input parameter of the function with their own values. The lookup on the `mValues` structure try to find in exist an appropriate tuple that match the `TOParameters` used in the present call. If it is implemented a precise lookup, the searched `TOParameters` must exactly match the `TOParameters` inside the data structure. Otherwise, for the memoized SLA lookup, the `TOParameters` provide a more relaxed lookup function in order to consider the SLA defined for our application.

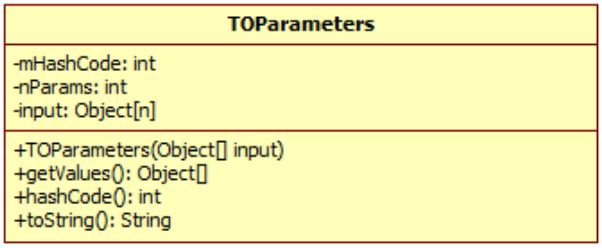


Figure 5.11: *The* `TOParameters` *class.*

5.5.2 Memory Allocation Policy

If more than one function needs to be memoized, available memory should be allocated to different functions so that the overall energy saving is maximized. We have to take into account the variability of typical parameter ranges - in financial applications, the statistic distribution of parameter values may evolve (slowly, compared with the execution time of the functions they are involved with), leading to suboptimal look-up tables or a need to periodic re-compilation. Our goal is to maintain efficient memoization tables across a long program life cycle, thus tables with memoized data should dynamically evolve. Memory is allocated to memoized methods in order to maximize the overall effectiveness of the approach. This is measured as the gain factor obtained for each function $G_i = 1/f_i$ weighed by the frequencies of occurrence of that specific function:

$$G_{tot} = \sum_i G_i f_i \tag{5.1}$$

where f_i is the frequencies of occurrence of function i.

We developed a memory calibration algorithm that defines the portion of memory to be assigned to each method that maximizes G_{tot}. This algorithm favors the methods that have a higher hit rate (α), based on the statistical distribution of their input parameters. Algorithm 2 allocates the available memory incrementally to each function by blocks of $S_{(d,i)}$ bytes. $S_{(d,i)}$ values are specific for each function i, according to the size of the input parameters and results to be stored for each function. The algorithm incrementally explores the possible allocation options of a new block of memory and identifies the function that would lead to the highest effectiveness gain if got assigned of that memory block. This is estimated by

Input: A set F of pure functions $f_i(x_{i,1}, \ldots, x_{i,n})$
Output: Memory percentages P_i for every function f_i
Data: $S_{d,i} = size(x_{i,1}) + \cdots + size(x_{i,n}) + size(f_i(x_{i,1}, \ldots, x_{i,n}))$
Data: The max memory S_M
Data: The reserved memory $S_{m,i}$ for f_i function (initially 0)
Data: $S_i = \Sigma_m S_{m,i}$
begin
  while $S_i < S_M$ **do**
    get $f_i \in F$ with max $\frac{G_i}{(S_{m,i}+S_{d,i})}$;
    $S_{m,i} = S_{m,i} + S_{d,i}$;
    update S_i
  end
  foreach $f_i \in F$ **do**
    update $P_i = \frac{S_{m,i}}{S_M}$
  end
end

Algorithm 3: Memory calibration algorithm.

$G_i = ((S_{(m,i)} + S_{(d,i)}))$, where $S_{(m,i)}$ is the memory already allocated to function i.

When the memoized application starts, the functions candidate for memoization are known as they have been previously identified by the pure function analysis module. Their frequencies of occurrence, and the mean and standard deviation values of their input parameters are estimated based on previous domain knowledge. A first run of the algorithm defines the initial allocation of the available memory space among the different functions. As long as the application runs, memory is allocated to memoized functions as the invocations happen, until the established portion of memory for each function is consumed. The trade-off module is invoked with a given frequency (β) in order not to create a time and power overhead. When it is invoked, first it runs the memory calibration algorithm with the new values of frequency of occurrence, mean and standard deviation. This leads to a new allocation of the available memory space. Then it eliminates the low-frequency entries in order to re-equilibrate the space according to the new allocation.

5.6 Conclusion

In this chapter, we present the architecture of Green Memoization Suite and analyze its main modules. The `Static pure function retrieval` module performs a static analysis of the bytecode of the application un-

der exam and, given a set of rules to identify the purity, it determines if a method could be considered pure or not. Given the pure function analysis results, the `Bytecode Modification` module execute the necessary instructions that permits to inject the code needed by the `Decision maker` meta-module. Finally, we illustrate the behavior of the `Memory Management` module that allow the use of the allocated memory to multiple pure-functions concurrently in order to correctly manage the SLA definitions.

CHAPTER 6

EXPERIMENTAL EVALUATION

This chapter presents and discusses the experimental results obtained from the application of the approaches described in Chapter 5. Section 6.1 presents the validation of the pure function re-engineering following the approaches introduced in Section 5.2 and 5.3. Section 6.2 proposes the evaluation of the two data structure (HashMap and RTree) used in our suite. Finally, in Section 6.3 are reported the Trade-off module validation and the overall memoization architecture evaluation.

6.1 Pure Function Re-Engineering Validation

Section 6.1.1 is dedicated to the validation of the bytecode analysis module implemented. Several benchmarks have been analyzed, and the results obtained have been compared to the manual examination of the correspondent source code. The overall results are encouraging, with an average precision of the result over 99% for all the applications considered. Section 6.1.2 focuses on the approaches proposed in Section 5.3, regarding how methods can be effectively modified to obtain their equivalent pure versions. The results obtained are still positive, with small impact related the overhead introduced following the re-engineering of the code.

The experiments have been conducted on an IBM eServer x3500 with two Intel Xeon Quad-core x5450 @ 3.00 GHz (12 MB L2 cache), FSB 1,333 MHz, 17GB PC2-5300 DDR2 SDRAM. All the tests have been conducted on Windows 2007, enterprise edition.

6.1.1 Bytecode analysis

As illustrated in Section 5.1, the analysis of the bytecode is the main challenge in terms of identification of pure functions. The proposed approach relies on the results of both the dependencies and purity analysis, performed as presented in Section 5.2 and 5.3. In order to test the precision of the results, three different sets of functions have been examined. Since the source code of all the classes is available, and the number of methods within each application is not prohibitive, a manual scan has also been performed, in order to determine the validity and precision of the results. The modules tested include in particular the generation of the tree related to the functions calls, the analysis of the data dependencies within a method, the identification of pure functions (or, if present, the specific sources of impurities).

The following list of conventions details the values presented. It has also been adopted for all the other benchmarks presented in the section:

- the **total methods** value includes all the methods declared in the considered package, and excludes all the functions belonging to the set of Java APIs;
- the value **purifiable** is evaluated as the percentage of solvable methods, on the total of non pure methods;
- the value **modifiable** is evaluated as the percentage of all the solvable and modifiable methods , on the total of non pure methods.
- the distinction between **pure** and **memoizable** methods has been introduced following the considerations made in Section 4.1.1. In fact, when a method is not pure only due its return type, it can still be memoizable if the visibility of such reference remains limited to the caller. If the value is not explicated, it is assumed equal to the number of pure methods;
- the **solvable** attribute refers to non pure methods that can be transformed in completely pure ones. For a detailed definition of such property, and the result following the relative modifications (see Section 4.1.1);

- the **modifiable** attribute refers to non pure methods that can be modified, in order to extract at least a pure code block from them. Typical scenarios involve functions that update the value of variables defined outside them (as illustrated in Section 5.3.2, 5.3.2);
- the **modifiable_obj** attribute refers to purifiable methods having the additional source of impurity related to the modification of arrays passed as parameters. The solver for this scenario has not been implemented yet. However, the concepts are very similar to the modification of global variables, as better explained in Section 5.3.2;

Non Linear Optimization Java Package

The `Non Linear Optimization Java Package` [2] is a set of mathematical functions, the major part resulting from a translation of the original FORTRAN versions; in particular, it implements routines that deal with unconstrained optimization problems, as well as resolutions of non linear equations. Table 6.1 and Figure 6.1 show the results, in terms of purity, of all the methods that compose the package.

Table 6.1: *Summarized analysis results for the* `Optimization` *library*

	Optimization
Total methods	70
Pure	5.71%
Purifiable	6.06%
Modifiable	33.33%

As observable, the library results poor from the point of view of purity for its methods; only four methods can be identified as pure, and other 4 functions can be effectively reconduced to a pure form, bringing the total percentage of pure methods up to 11.38%. However, a consistent increment can be experienced if considering the resolution of methods that are non pure due the modification of arrays, passed as parameters. Considering these functions, the percentage of methods that can be modified from the proposed framework increases to an encouraging 33.33%. The manual examination of the source code has not produced altered results, determining an accuracy of 100% in terms of identification of pure functions.
Table 6.2 shows the distribution of the sources of impurities, in terms of frequency with which they appear (at least once) in each method of the library; the detailed explanation for each flag is reported in Section 5.2.2. As a direct consequent of the previous observation, more than 60% of the methods

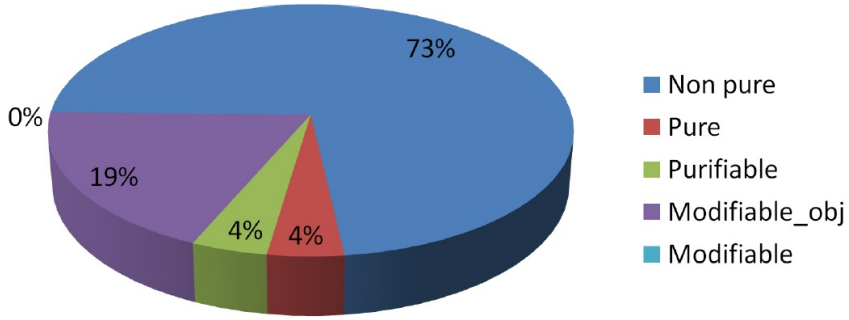


Figure 6.1: *Purity analysis results for the* `Optimization` *library*

are not pure due the modification of array parameters. The other higher contributes are, respectively, the invocation of non pure methods and the presence of objects in the signature. The first, however, could include calls to methods that can be purified, while the second kind of impurity can be solved, for example, following the resolution for static or non static global variables. The results show again the good margin of operation possible for the library, in terms of bytecode re-engineering.

Table 6.2: *Distribution of the sources of impurity for the* `Optimization` *library*

	Optimization
Signature	41.43%
Return type	1.43%
Static read	25.71%
Static write	0.00%
Fields read	10.00%
Fields caller	2.86%
Fields write	0.00%
Objects	62.86%
Other	0.00%
Invocations	64.29%

JOlden

The `JOlden` suite is a set of methods derived directly from the original `Olden` suite. Written in C, this group of applications has been used by Martin Carlisle and Anne Rogers to evaluate a system that parallelizes programs with dynamic data structures. The original version relies upon parallel constructs, while the Java one (translated by the member of the Architecture and Language Implementation [14]) is executed in a sequential approach. The list of the applications included in the suite, along with a brief description for each one of them, is reported in Table 6.3.

Table 6.3: *List of benchmarks of the* `JOlden` *suite.*

Benchmark	Description
BH	BarnesHut Nbody algorithm
BiSort	Bitonic sorting algorithm
Em3d	Electromagnetic waves simulation
Health	Healthcare system simulation
MST	Bentley's algorithm for the minimum Spanning Tree of a graph
Perimeter	Evalaute the perimeter of an image represented as a tree
Power	Maximize the economic efficiency of a community
TSP	Randomized algorithm for the Travelling Salesman Problem
TreeAdd	Recursive algorithm for the sum of the nodes' value of a tree
Voronoi	Voronoi's diagram for the realization of a random set of points

Figures 6.2, 6.3 and Tables 6.4, 6.5 summarize the results obtained from the purity analysis of all the distinct benchmarks. The overall percentage of pure methods is quite low, ranging from a 25% for the `Bisort` application (which is, however, composed only from 12 methods) to 0% for other benchmarks; when considering the entire suite, the pure methods can account for only 5.8% of the total. Instead, the results related to the number of purifiable methods are a bit more encouraging, averaging a 13.01% overall. Clearly, this figure refers only to the absolute number of methods satisfying such property, without any additional analysis to determine the effective benefits that could be obtained by such modifications (performed, instead, in Section 6.1.2). Considering also the methods that can be solved, at least partially, in terms of purity, the results look even better, settling to a total of 31.06%. The value related to non pure methods, due the modification of array parameters, is not as relevant as for the considered optimization library, accounting for a total of 5 methods, all defined within the `Power` benchmark.

The manual examination of all the benchmarks included in the suite con-

Table 6.4: *Summarized analysis results for the* `JOlden` *benchmarks (part 1)*

	BH	BiSort	Em3d	Health	MST
Total methods	67	12	17	20	29
Pure	2.98%	25.00%	0.00%	5.00%	10.34%
Purifiable	16.42%	0.00%	11.76%	5.26%	11.54%
Modifiable	42.42%	22.22%	23.53%	26.32%	30.77%

Table 6.5: *Summarized analysis results for the* `JOlden` *benchmarks (part 2)*

	Perimeter	Power	TSP	TreeAdd	Voronoi
Total methods	38	23	6	13	67
Pure	13.15%	0.00%	0.00%	0.00%	4.48%
Purifiable	15.15%	13.04%	16.67%	23.08%	17.19%
Modifiable	18.18%	56.52%	33.33 %	30.77%	26.56%

firms the obtained results, with one exception related to the `Em3d` application. In the class `Node` the method `void updateFromNodes` modifies an object declared internally; however, the address of a global variable is assigned directly to it, thus referring on an object defined outside the scope of a method. Hence, the update of such value determines a side effect for the application in execution, and can not be completely solved in terms of purity. However, it is still possible to apply bytecode modifications to ensure that, at least a part, can be considered pure (the process is the same as illustrated in Section 5.3.2). The results of such application should hence account one purifiable method instead of two, and three modifiable methods instead of two. Even with this discrepancy, the implemented analysers can still manage to correctly identify 99.66% of the total functions that compose the suite.

The distributions of sources of impurities (shown in Tables 6.6, 6.7 present various results, strictly related to the kind of application which they refer to. For example, it is evident how the `BH` benchmark is heavily impacted by operations on global variables (in both read and write). Moreover, almost 40% of the total functions return an object, not admissible in terms of purity. The largest part is in fact constituted by methods that operate on custom-defined vectors, and return such kind of variable. Similar situations are experienced by the `Perimeter`, `TreeAdd` and `Voronoi` benchmarks, each one of them having more than 50% of the methods presenting such characteristic.

Operations on global static and non static variables are very common for the set of selected benchmarks. In fact, an average higher than 50% of the

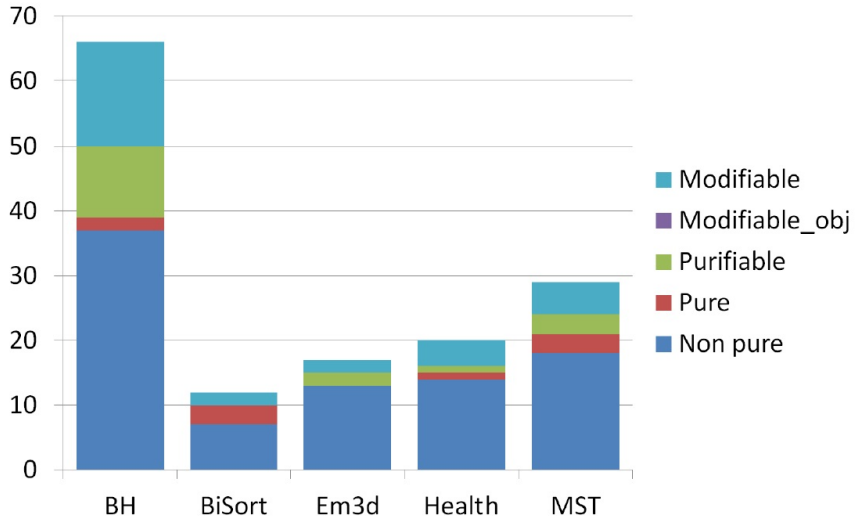


Figure 6.2: *Purity analysis results for the* `JOlden` *benchmarks (part 1)*

total methods perform operations of reading fields, usually belonging to the calling object, but also on objects passed as parameters to the method. The reading access to static variables is another common factor for some of the applications. For example, for `BiSort`, `Em3d`, `Perimeter` and `TreeAdd`, such value accounts more than 30%. Less common are the update of values related to fields, static variables and arrays passed as parameters. Only `BH`, `BiSort` and `Perimeter` present results of at least 10% for impurities related to fields, with the latter as the only one that overcomes the 10% threshold for operations of writing on static variables. The impurities related to the modifications of arrays passed as parameter are not significative for this set of benchmarks, with only the `Power` and `Perimeter` applications holding statistical importance (more than 10% of the total methods).

Java Grande

The Java Grande benchmark suite [1] is a set of applications designed to provide ways of measuring and comparing alternative Java execution environments, in ways which are important to `Grande` applications. A `Grande` application is one which uses large amounts of processing, I/O, network bandwidth, or memory. This set include not only applications in science and engineering, but also corporate databases and financial simulations. The sequential benchmarks of Java Grande (Section 3), related to

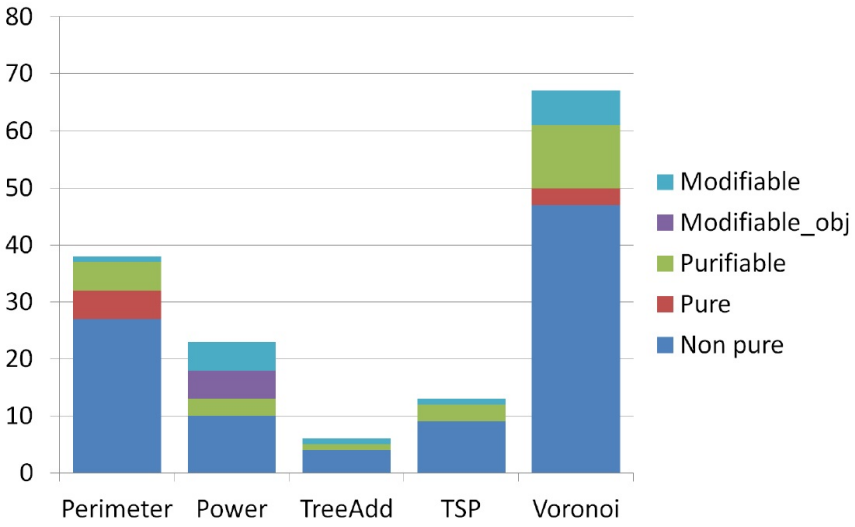


Figure 6.3: *Purity analysis results for the* `JOlden` *benchmarks (part 2)*

large scale applications, have been considered. Table 6.8 describes briefly the applications used to verity the bytecode analyzers implemented.

Figure 6.4 and Table 6.9 summarize the results obtained from the purity analysis of the presented benchmarks. As observable, the methods that can be considered pure are only a minor part (1.69%) for the `MonteCarlo` benchmark, while the others have not even a single pure function implemented. The results are quite better if considering the methods that can be manipulated in order to obtain an equivalent, pure version. `MonteCarlo` presents again the higher contribution (25.99%), while the average settles

Table 6.6: *Distribution of the sources of impurity for the* `JOlden` *suite (part 1)*

	BH	BiSort	Em3d	Health	MST
Signature	52.24%	33.33%	23.53%	45.00%	41.38%
Return type	38.81%	8.33%	41.18%	40.00%	34.48%
Static read	10.45%	33.33%	35.29%	15.00%	17.24%
Static write	5.97%	8.33%	11.76%	5.00%	10.34%
Fields read	76.12%	41.67%	58.82%	70.00%	51.72%
Fields caller	40.30%	33.33%	23.53%	30.00%	27.59%
Fields write	10.45%	25.00%	0.00%	5.00%	0.00%
Objects	4.48%	0.00%	0.00%	0.00%	0.00%
Other	7.46%	0.00%	5.88%	0.00%	0.00%
Invocations	53.73%	50.00%	58.82%	55.00%	37.93%

Table 6.7: *Distribution of the sources of impurity for the* `JOlden` *suite (part 2)*

	Perimeter	Power	TSP	TreeAdd	Voronoi
Signature	42.11%	21.74%	50.00%	30.77%	37.31%
Return type	55.26%	17.39%	50.00%	38.46%	52.24%
Static read	39.47%	13.04%	50.00%	30.77%	8.96%
Static write	18.42%	4.35%	16.67%	7.69%	4.48%
Fields read	36.84%	69.57%	16.67%	46.15%	43.28%
Fields caller	15.79%	43.48%	16.67%	23.08%	7.46%
Fields write	13.16%	0.00%	0.00%	0.00%	1.49%
Objects	13.16%	21.74%	0.00%	0.00%	1.49%
Other	13.16%	0.00%	0.00%	0.00%	0.00%
Invocations	50.00%	43.48%	50.00%	76.92%	52.24%

Table 6.8: *List of benchmarks of the* `Java Grande` *suite, Section 3, used.*

Benchmark	Description
Euler	Computational Fluid Dynamics
Moldyn	Molecular Dynamics simulation
MonteCarlo	Monte Carlo simulation
Ray Tracer	3D Ray Tracer

on a value of 12.58% of the total. However, as presented in the following section, the number of purifiable methods for the `MonteCarlo` benchmark does not effectively reflect the impact that it has on the global application, mainly due the nature of the method themselves. Considering instead the results relative to the modifiable methods, the values are more encouraging; each application reaches at least the 25% of the methods that compose them, while `MonteCarlo` presents a spike again, up to 62.64%. At a first glance, hence, bytecode modification looks not only possible, but also that could provide some impact on the overall applications.

Table 6.9: *Summarized analysis results for the* `Java Grande` *suite, Section 3*

	Euler	Moldyn	MonteCarlo	Ray Tracer
Total methods	35	27	63	177
Pure	0.00%	0.00%	1.69%	0.00%
Purifiable	14.29%	3.70%	25.99%	6.35%
Modifiable	28.57%	33.33%	62.64%	25.40%

The manual examination of the source code of the applications confirms the results obtained for all the benchmarks, with the exception of `Euler`. Due to the complex expressions present in two methods within the `Tunnel`

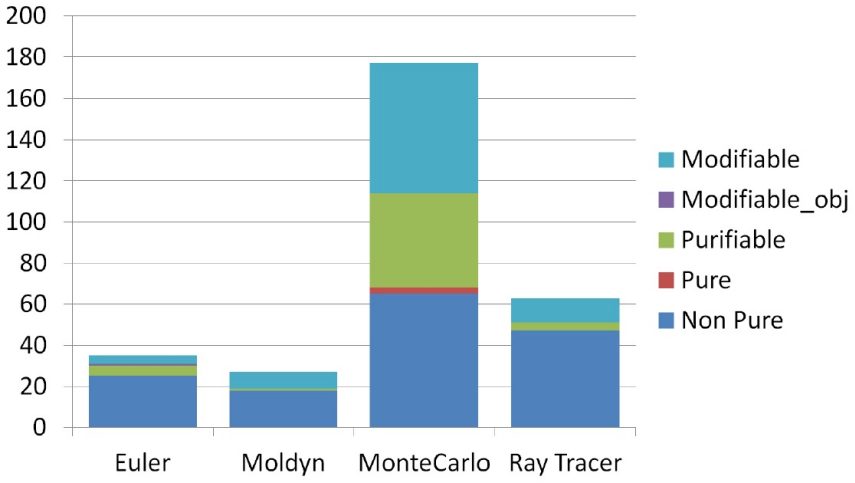


Figure 6.4: *Purity analysis results to the* `Java Grande` *suite, Section 3*

class, the analyser can not identify the `objectref` related to the operation on the matrix field, which is however the calling object. Hence, since this is the only code portion that present such impurity, it fails to individuate the update of a global variable as a source of impurity for both methods. The correct results would have two more methods in the modifiable group, and only three methods as purifiable (resulting in a decrease from 14.29% to 8.57% for such percentage). Even with this discrepancy, the implemented analysers can correctly identify 99.34% of the total functions that compose the suite.

The distribution of sources of impurities (shown in Table 6.10) underlines the reading from local variables (especially non static ones) as one of the major cause of impurity. Each method presents this characteristic for at least 50% of its methods, up to 74.60% for the `Ray Tracer` application. Such benchmark, however, presents also the highest value in terms of function having an object as return type; 38.10% of its methods have such property, that, as already explained, it is usually a consistent obstacle in terms of purity. Other significant results include the number of functions that read from static variables (e.g. `Moldyn` has almost 60% of its methods presenting such behaviour), while the update of such values is far less common (each benchmark has below 10% of the total methods with such characteristic, with the exception of the `Moldyn` application). The impurities related to the modifications of arrays passed as parameter are not numerically significative for this set of benchmarks.

Table 6.10: *Distribution of the sources of impurity for the* `Java Grande` *suite, Section 3*

	Euler	Moldyn	MonteCarlo	Ray Tracer
Signature	42.86%	18.52%	20.34%	55.56%
Return type	14.29%	0.00%	11.86%	38.10%
Static read	31.43%	59.26%	14.12%	22.22%
Static write	0.00%	25.93%	1.13%	3.17%
Fields read	71.43%	55.56%	53.11%	74.60%
Fields caller	28.57%	51.85%	44.63%	34.92%
Fields write	8.57%	0.00%	0.00%	3.17%
Objects	5.71%	0.00%	0.00%	3.17%
Other	0.00%	0.00%	0.00%	3.17%
Invocations	57.14%	66.67%	30.51%	65.08%

6.1.2 Bytecode modification

The `Bytecode modification` module has been extensively tested in order to guarantee the correctness of the produced results. Necessary condition for the application of the bytecode re-engineering approach is the fact that the new versions have to behave exactly as the original one, in terms of produced effects. The property has been tested on a number of functions, both ad-hoc generated and belonging to the considered benchmarks, showing no discrepancy between the two versions, and hence confirming the goodness of the implementation. Equally important is the evaluation of the overhead that such modifications could introduce. In fact, the typical operations performed by the solver presented in Section 5.3 are responsible for bytecode transformations that could results in less-efficient versions (in terms of time of execution). For example, the introduction of an additional function call generally slows down the execution, as well as the saving in temporary variables for values needed by the caller. Since one of the main aspects of the proposed work is to enable non pure functions to be memoized, through the use of such transformations, a small overhead is generally considered acceptable, while higher values could not be effectively balanced by the application of the memoization technique. In any case, the effects on the global application should be evaluated with a run time profiling, to determine wherever the re-engineering of the considered methods produces effectively a benefit (in terms of both execution time and energy saving).

Optimization

Since the package is not a benchmark application, but a set of mathematical functions, it is not possible to consider the global impact of the proposed modifications; however, it can be useful to evaluate the behaviour of the single functions, with respect to the original versions.

The set of functions selected for the first test are the following:

- `f_to_minimize` of the `FminTest` class;
- `fmin` of the `Fmin` class.

Summarily, the first one is not pure due the access to fields of the caller (only in reading), while the latter is not implicitly pure since calls various time such function within its execution; moreover, its signature contains an object, used as caller for `f_to_minimize`.
The designed solver is the `FieldsReadOnlySolver`, that loads on the stack the required fields, and isolates the pure code block within another, pure method. The overhead introduced by such operation is practically negligible, since the execution time increases only by 0.07% (attributable mainly to the new invocation inserted). The behaviour of the calling `fmin` function does not really reflect such result, since the execution time experiences an overall decrease of 0.33%; however, given the dimension of such variation, it can be classified under environmental uncertainty.
The second step of resolution is the direct call, for `fmin`, of the pure function equivalent to `f_to_minimize`. This operation, which is performed by the `CalledMethodSolver`, introduces the source of impurity relative to the access to global variables, but eliminates the calls to the original, non pure, `f_to_minimize`. The execution time, after such modification, is almost the same as the original one; this is expectable, since the number of invocations has remained the same.
To solve the last impurities, it is necessary to execute the `FieldsReadOnlySolver` again; this will wrap the pure part of the `fmin` function in a separate method, while the external one will manage the load of required fields. The insertion of an additional call impacts as a negligible 0.23% on the total time of execution. Globally, the sequence of transformations does not introduce a consistent overhead (with respect to the original versions) and hence can be accepted, regardless the effective application of the memoization. The described results are summarized in Table 6.11.

The second set of functions selected for testing are the following:

- `f_to_minimize` of the `UncminTest_f77` class;

Table 6.11: *Summarized overhead results for the* `Optimization` *library (part 1); the* T_exec *values reported refer to the execution of the functions* 10^7 *times.*

Method	T_exec (ms)	Variation	Notes
`f_to_min`	5516.2		Original
`f_to_min1`	5520.2	0.07%	After modif. 1
`fmin`	6309.6		Original
`fmin`	6288.7	-0.33%	Original (calling `f_to_min1`)
`fmin2`	6306.5	0.05%	After modif. 2
`fmin3`	6323.9	0.23%	After modif. 3

- `fstocd_f77` of the `Uncmin_f77` class.

Conceptually, the methods behave similarly to the previous pair of functions; the `f_to_minimize` function, in particular, is very similar to the homonymous one. The second method presents impurities related to the signature, the modification of arrays passed as parameter, and clearly the internal calls to the non pure `f_to_minimize` function. The steps of the resolution for this scenario are very straightforward to those described in the current section: a first modification to extract the pure part from `f_to_minimize`, the implementation of the direct calls to it from `fstocd_f77`, and the extraction of the fields required for such method. The function `fstocd_f77` will not still result pure, due the updates operated on the cited arrays, but such additional resolution could be eventually implemented, depending on the calling application in which it is used. The results associated to such operations are reported in Table 6.12, following the conventions presented before. Similarly to the previous case, the overhead values are quite encouraging, with a maximum of 2.27% for the final version of the `fstocd_f77` method. However, when deciding to not solve the impurities related to arrays, it is more convenient to stop the resolution process at step 2. This avoid the insertion of an additional call, that will not make anyway the method completely pure. In this case the overhead is still more restrained (below 0.50%).

JOlden

One of the advantage of operating with the benchmarks of the `JOlden` suite resides in the fact that such applications are developed with a small number of methods; hence, the modification of even a little number of functions should provide a notable impact on the entire benchmark. From a set of such applications, a subset of purifiable methods has been considered and solved (based on observations presented more in detail in the following

Table 6.12: *Summarized overhead results for the* `Optimization` *library (part 2); the* T_exec *values reported refer to the execution of the functions 10^7 times.*

Method	T_exec (ms)	Variation	Notes
`f_to_min`	4525.4		Original
`f_to_min1`	4448.5	-1.70%	After modif. 1
`fstocd_f77`	13259.5		Original
`fstocd_f77`	13261.5	0.02%	Original (calling `f_to_min1`)
`fstocd_f772`	13316.7	0.43%	After modif. 2
`fstocd_f773`	13561.0	2.27%	After modif. 3

section).

BH The `BH` benchmark is one of the biggest of the suite, in terms of number of methods; nonetheless, only two functions out of 67 are identified as pure. Not every method, of the twelve flagged as purifiable, is suitable for the transformations proposed. However, a subset of good candidates has been individuated and re-engineered, in accordance with the proposed approach. The following list presents each method considered, with considerations about the relative sources of impurities, along with the transformation that have been performed upon it.

- **MathVector.value**: the method is basically a getter for an element of global array. Usually, getter for fields of primitive type should not be considered for modification; however, the present function is called within other methods, and constitutes an element of impurity. Hence, an alternative is to implement the getter `getData()`, which returns the entire array, and then retrieve from it directly the desired element. While this solution does not impact the `value` method, it will be useful when dealing with the other functions that call it;
- **MathVector.dotProduct**: this is a good example of application of the previous solution. This method executes a loop in which it retrieves the elements of the interested (global) array, through the call of the non pure function `value`. Hence, such array can be passed to it as parameter, through the use of the previously defined `getData()`. The resolution is assimilable to the one performed by the `Fields-ReadOnlySolver`, dealing instead with non primitive fields;
- **MathVector.absolute**: the method is very similar to the previously descripted `dotProduct`, refer to it for further details on the modifications applied;

- **MathVector.distance**: the method is very similar to both `dotProduct` and `absolute`; however, it also reads from the fields of an object passed as parameter. The pure method, hence, will receive as input both the required fields, using the `getData` defined function;
- **Node.oldSubIndex**: it calls the original `value` function, which is non pure. It can, however, be modified in order to call directly the pure equivalent of such method, providing the field array as input. Moreover, it reads from the field of an object parameter. This is addressed as described for the `distance` method;
- all other methods calling `dotProduct`, `absolute`, `distance`: regardless of the other sources of impurities for such methods, the calls to the three cited methods can be substituted with the correspondent one to the equivalent, pure methods. This operation should not introduce consistent overhead, since the number of calls to functions remains unchanged (this consideration is valid for every other benchmark considered in the current section);
- all the other methods flagged as purifiable are not considered, since they can be assimilated to getter and/or are not relevant for the purification of other functions(this consideration is valid for every other benchmark considered in the current section).

The result associated with such operations are encouraging. For both size of benchmarks considered (A,B), respectively with inputs of 10^5 and $5*10^5$, not only the overhead introduced does not affect the execution times, but it even slightly improves them (the detailed results are shown in Table 6.13). Hence, it is possible to assume that the modifications described will result in minimal impact on the overall performance of the application, if not even in a small improvement. Ideally, the application of the memoization should further improve such results, determining a consistent impact overall.

Table 6.13: *Overhead results for the `BH` benchmark; the modifications of the select methods result in even a slight improvement of the execution time.*

	T_exec (ms)	Variation
Original (size A)	21207	
Modified (size A)	20452	-3.56%
Original (size B)	426068	
Modified (size B)	412336	-3.22%

Em3d The `Em3d` benchmark has only one method that can be purified; however, since it is assimilable to a getter, and it is not determinant for the impurities of other methods, its resolution is not suggested. From the set of modifiable methods, instead, one method can be a good candidate (the others are basically setters).

- **Node;computeNewValue**: the method presents a loop in which it accesses two global entities, an array of `double` and an array of `Node` (class defined within the benchmark). The latter, moreover, access to another global variable, declared within it, adding hence additional complexity to the sequence of reads. The implemented solution deals with the first array as already illustrated, and with the second through an additional operation. The wrapping method needs to initialize such values, accessing all the needed variables and storing them in a new array; then, such data structure is used in the calling of the (now) pure method, along with the other array and the original parameters. In this way, the code block that effectively performs the computation results pure.

As expectable, the overhead related to the re-engineered method accounts not only the additional call to the pure method, but most of all the operations required to instantiate the array of parameters needed. The proposed solution has been tested providing the benchmark with inputs of size A and B, respectively of <10^5, 10, 100> and <10^5, 10, 1000> (format corresponding to <number of nodes, out-degree of-each node, number of iterations>). The results (Table 6.14) shows effectively an increment of the time required for the execution between 13% and 15%; this overhead can not be considered negligible, and else additional profiling has to be taken in account. A solution could be the analysis of the behaviour of the benchmark, after the application of the memoization; if the method `Node;computeNewValue` is called frequently with the same values (for the arrays interested by the modification process) the overall result could be a benefit for the application, otherwise the original version is preferable.

Perimeter The `Perimeter` application does not offer much in terms of purifiable methods. Only two among them look suitable for effective re-engineering, in order to extend the set of pure functions.

- `QuadTreeNode;checkOutside`: the method performs small computation, and returns a value based on some comparisons. Such operations involve two static variables, and hence can be addressed with the

Table 6.14: *Overhead results for the `Em3d` benchmark; the modifications of the select method result in a not negligible overhead on the execution time of the application.*

	T_exec (ms)	Variation
Original (size A)	8674	
Modified (size A)	9866	13.74%
Original (size B)	71684	
Modified (size B)	81929	14.29%

procedure illustrated in Section 5.3.2. While the method itself does not benefit in an impressive way due such modifications, other functions calling it can now refer to the pure, memoizable version.

- `QuadTreeNode;checkIntersect`: this function is a direct example of the benefits obtained with the previous operation. The method is a sequence of invocations to the `checkOutside`, and presents no other source of impurity. Hence, substituting such calls with those referring to the new, pure version of the function, the method `checkIntersect` results now pure from this point of view. The additional impurity introduced by the operation (the presence of static variables in the stack for such invocations), is addressed in the same way illustrated for the previous case. Any other call to both methods considered can now be substituted with the one correspondent to the pure versions.

The application has been tested with a fixed input of 10^6 (number of levels in the quadtree), cycled respectively 10^8 times for size A, and 10^9 times for size B. The results, shown in Table 6.16, show an almost negligible impact of the overhead introduced, in terms of overall performance. In the limits of the environmental noise present, it looks like that the modifications can translate even in a slighter improvement; in fact, only considering the `checkIntersect` method, at each execution of such function the JVM can save an average of ten `GETSTATIC` instructions, substituted instead by local `DLOAD` to load the required values.

Power All the methods flagged as purifiable, for the `Power` benchmark, can be effectively modified, since there are not getters among them. In particular, the list of such functions includes:

- **Demand;findG**: this method performs very small computation, but it involves the reading of two static variables. Hence, the normal reso-

Table 6.15: *Overhead results for the* `Perimeter` *benchmark; the introduced overhead does not impact in a significative way on the execution time of the application.*

	T_exec (ms)	Variation
Original (size A)	21192	
Modified (size A)	20544	-3.06%
Original (size B)	206319	
Modified (size B)	207749	0.69%

lution can be applied, and the new, pure method implemented in the class;

- **Demand;findH**: this method behaves similar to the previous one. The resolution process is also practically the same;
- **Root;reachedLimit**: the computation performed by this function involves the use of several field variables, defined to the calling object. Among them, two are of primitive type (`double`), while the others refer to fields of type `Demand`, on which another `double` field is retrieved. The pure method, hence, will have all the four required `double` fields as parameters, while the wrapping one will retrieve such values and load them on the respective stack. Since such variables are defined as `protected`, all other methods outside the current class must retrieve them through the self-implemented getters (and not directly).

Even if the modified methods result small in terms of computation, the values obtained by the tests performed are encouraging. In fact, the overhead introduced accounts for an aggravation, in terms of time required by the execution, lower than 1% for both input sizes. In other terms, the performance of the original and the re-engineered version are practically the same, with the difference that the latter allows additional, presumable benefits following the application of memoization. Size A of the benchmark has been performed in correspondence of 10 times the normal execution, while size B iterated the elementary case up to 100 times.

TSP The `Travelling Salesman Problem` benchmark is very small in terms of number of methods; however, three out of 13 of them (corresponding to a notable 23.08%) are flagged as purifiable, and effectively are good candidates. The considered methods are summarized in the following list:

Table 6.16: *Overhead results for the* `Perimeter` *benchmark; the introduced overhead does not impact in a significative way on the execution time of the application.*

	T_exec (ms)	Variation
Original (size A)	5488	
Modified (size A)	5479	-0.16%
Original (size B)	54290	
Modified (size B)	54693	0.74%

- **Tree;distance**: this method operates using four different global variables. Two of them are related to the calling object, while the other two are referred to the object of type `Tree` that the method has as input. The method can be purified, translating externally all the impurities, as illustrated in Section 5.3.2. The resulting method will call a new, pure function, providing the required values on the stack;
- **Tree;median**: the function presents an impurity related to the presence of a random-generated value within it. That is accomplished by the function `Random;nextRand`, which is belonging to the set of Java APIs. However, since such variable does not have any kind of dependencies towards other values, the call to the non pure method can be extracted. The resulting pure method will have an additional parameter as input, corresponding to the random-generated value; such variable will be provided on the stack by the original method, using the same `Random;nextRand` function to create it;
- **Tree;uniform**: the characteristics of this method are very similar to those described for the previous one. Refer to the`Tree;median` description for additional details.

The results obtained from the two sets of measurements are very interesting. Overall, for both tests (size A corresponding to 10 iterations, size B to 50, each one of them for an input of 10^6 number of cities), an improvement higher than 10%, in terms of execution time, has been experienced (Table 6.17.

Since the modifications relative to the `Tree;median` and `Tree;uniform` can not be responsible for such variations (the number and type of instructions performed are practically the same), the reason for such consistent improvement should be located in the re-engineering of the `Tree;distance` function. The original function reads from four different global variables, two times for each; hence, the new method will reduce by half the number of `GETFIELD` (and correspondent `ALOAD`) instructions required, with the

Table 6.17: *Overhead results for the `TSP` benchmark; the re-engineering of the considered functions determines an overall, average improvement of more than 10% for the benchmark.*

	T_exec (ms)	Variation
Original (size A)	23112	
Modified (size A)	20570	-11.00%
Original (size B)	115711	
Modified (size B)	103400	-10.64%

only `DLOAD` now needed. Assuming that the loading instructions take practically the same time to be executed, at each invocation four less `GETFIELD` are required. Using a smaller input, and considering a version of the benchmark in which only the `Tree;distance` has been modified, the time saving has still a value around 10.11% (correspondent approximately to 27*ms*), confirming the above observation. Moreover, with the analysis tool `perf4j` [3], it can be noticed that such function has been executed 10030070 times in this scenario. Hence, that will be exactly the number of `GETFIELD` instructions that the JVM does not need any more to perform. Correlating the values, it is possible to state that the reduction of around 10^6 `GETFIELD` instructions translates in approximately 2.69*ms* not required by the execution anymore. For more complex functions, this approach could also be used to provide an estimate of the impact that the re-engineering method can have on the overall application; in this case, this is not significant, since the `Tree;distance` function performs very small computation, which would require more accurate tools for the analysis.

Voronoi Along with the first benchmark considered, `Voronoi` is the biggest application, in terms of number of methods implemented, of the `JOlden` suite. Following the purity analysis, 11 out of 67 methods have been flagged as purifiable; however, only five of them can be effectively modified (the others being only getters).

- **Vec2;cprod**: the method reads from global variables, two belonging to the calling object and the other two referring to the only parameter. The resolution is approached in the same way as described for other methods considered in the current section;
- **Vec2;dot**: the method is very similar to the already considered `Vec2;cprod`, and the resolution performed exactly the same;
- **Vec2;magn**: the method is similar to those considered, with the dif-

ference that it only reads from two fields of the calling object. The resolution is however following the same approach;

- **Vertex;ccw**: the impurities related to this function are the same as considered for the other methods described. In this case, the original function has two object as parameters, from which it read two fields each. Hence, the pure function will have a total of six parameters, corresponding to the six values required by the computation;

- **Vertex;incircle**: the impurities of this function constitute an extension of the `Vertex;ccw` case. In fact, not only the method has three object as parameters, but from each of them (including the caller) it reads three different fields. The resolution process is the same as described for the previous method, and the new pure method will need twelve parameters to be correctly executed.

The tests, executed with input of 10^5 nodes, confirm the results obtained from the other applications of the suite. For both case executed (size A corresponding to 20 loops, size B corresponding to 50 loops), the average overhead introduced accounts no more than 1.50%, as shown in Table 6.18. Unlike the `TSP` case, there is not a consistent saving in terms of `GETFIELD` or `GETSTATIC` instructions, but the resulting version is almost equal, in terms of time needed for the computation, with respect to the original one; however, since it allows the application of memoization, it is expectable that the overall results will be better. The only concern is related to the space of parameters; for the last method considered, the pure equivalent requires twelve values as input. That could result prohibitive in terms of memory needed by the result table, but this can only be determined only following a run time profiling of the application.

Table 6.18: *Overhead results for the `Voronoi` benchmark; the overhead introduced by the re-engineering of the considered methods accounts for no more than a 1.50% increase of the time of the execution.*

	T_exec (ms)	Variation
Original (size A)	22685	
Modified (size A)	22954	1.19%
Original (size B)	55774	
Modified (size B)	56599	1.48%

Java Grande

The applications of this set are generally more computationally expensive than those present in the `JOlden` suite. Hence, testing the proposed approaches on them could potentially lead to different results, with respect to those obtained in the previous sections. Of the four application tested in section 5.1.3, only two have been selected (`Euler` and `Ray Tracer`), while the others do not have effective margin to propose modifications. In particular, even with almost a 26% of methods identified as purifiable, the `MonteCarlo` benchmark is not suitable for re-engineering, since all the methods flagged as such are in reality only getters.

Euler Two methods have been selected for the tests regarding the `Euler` application:

- **Vector2;dot**: this function performs small computation using two fields of the calling object. The resolution is straightforward with respect to the same scenario analysed in the section;
- **Vector2;magnitude**: also for this function, the impurities are related to the access on global variables (this time also of the only parameter). The new pure method will have a total of four values as input, up from the original two.

The benchmark provide two set of data as inputs, respectively size A and size B. The results (Table 6.19) show a small increase on the overall performance, ranging from -0.48% (size A) to -2.75% (size B). Examining the code of the modified methods, the `Vector2;dot` function can be identified as the main contributor to such values. In fact, during its computation it reads two times from the same fields, while the re-engineered version halves the number of `GETFIELD` required. The `Vector2;magnitude` method, instead, performs only one access to each of the field considered in the resolution, and hence does not provide much impact under this point of view.

RayTracer Only one out of four total methods flagged as purifiable is not a getter, and hence can be a target for re-engineering. However, it can be useful to observe the behaviour of the application, in terms of performance, following the modification of just a method within it.

- **Vec;dot**: this method operates on two custom defined vectors, that constitute its set of parameters. The computation it performs involves

Table 6.19: *Overhead results for the `Euler` benchmark; the re-engineering of the considered functions determines an overall, average improvement of about 1.5% for the benchmark.*

	T_exec (ms)	Variation
Original (size A)	2696	
Modified (size A)	2683	-0.48%
Original (size B)	5672	
Modified (size B)	5516	-2.75%

the access to three fields of such objects (corresponding to the three spacial coordinates), for a total of six reads. The resolution is delegated to the already described `FieldsReadOnlySolver` class, which will define the new pure method. Clearly, all the calls to the original `Vec;dot` will then be redirected to the new, pure function, through the action of the `CalledMethodSolver` class.

Similarly to the previous results, the overhead introduced is very modest and can be assimilated to environmental noise. As shown in Table 6.20, in fact, the average decrease in performance can be estimated in around 1.60%, an acceptable value when considering the enabling which now the benchmark has. A run time profiling will be needed if considering the choice to re-engineer such method in order to apply the memoization.

Table 6.20: *Overhead results for the `Ray Tracer` benchmark; the overhead introduced by the re-engineering of the considered method accounts for no more than a 1.60% increase of the time of the execution.*

	T_exec (ms)	Variation
Original (size A)	2944	
Modified (size A)	2970	0.88%
Original (size B)	32354	
Modified (size B)	33134	2.41%

6.2 Data Structure Validation

In this section, we show prove how memoization approach works good with R-tree structure and we validate it measuring power consumption of test functions as well as our use case functions.

6.2.1 Settings

The aim of choosing the memoizable functions is observing the energy consumption in the following steps:

- Original function execution;
- Function execution by adding memoization and Trade off block;
- Function execution by adding memoization and Trade off block, using precision.

We used 512MB of the RAM memory available for the tests to verify the R-tree structure within memory management. The data are obtained by measuring the energy consumption of the server during the stress tests. The tests are done in three phases, the generation and saving the inputs of the functions, execution of original function and, lastly, execution of memoized functions.

We measured power consumption through a specific measurement kit. The current absorbed by the system is measured by means of an ammeter clamp. Ammeter clamps have a Hall current sensor inside and allow non intrusive measures. The analog signal acquired by the ammeter clamp is processed by a NI USB-6210 DAQ (Data Acquisition Board) that is interfaced via USB with a server different from the system that executes the workload. The sampling frequency employed is 250 MHz. Data acquisition is performed by a software tool developed ad hoc for this study using LabVIEW. The tool acquires and stores samples of current energy consumption every 4 microseconds (i.e., with a sampling frequency of 250 MHz). We automatically evaluated the execution time of each workload by sensing the difference of power absorption w.r.t. the idle state.

To evaluate the general approach, we have selected `Implied Volatility`, `Black and Scholes`, `Fourier Series` and `Unicredit` functions. We have done the phase of memory allocation with 1 million of data. For `Fourier series` the allocation is done with 100.000 data.

6.2.2 Evaluation

Based on the performance evaluations on the financial functions and the mortgage program, we sometimes reach a good result, but sometimes not. The reasons of bad results are the followings:

- R-tree method is a big structure for complicated systems. If a function is simple to compute, its performance may deteriorate. So, Hash tables give better results than R-tree structure.

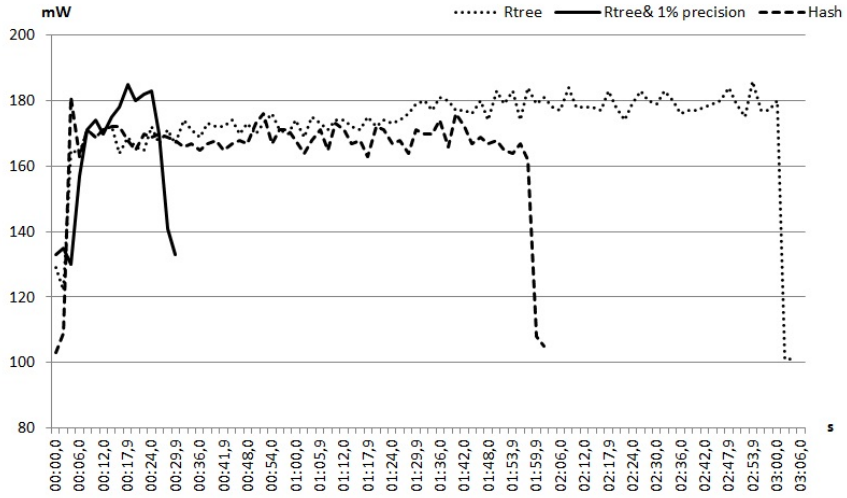


Figure 6.5: *The energy consumption and timing graph for Fourier Series*

- The important parameters depend exponentially form the number of dimensions. Therefore, R-tree so far operates efficiently if the number of dimensions is fairly small.

We obtained different results on R-tree structure and Hash table in the financial application tests. Without considering error rate on the inputs, we reached for Black and Scholes 51 times, for Implied Volatility 14.6 times and for Fourier Series 1.56 times worse computation time. Then, giving 1% of error on each input of every function allows us to retrieve for Black and Scholes 51 times worse, for Implied Volatility 11 times worse and for Fourier Series 5.22 times better computation time. Increasing the precisions bring better results at all.

On the other hand, the situation alters on the mortgage functions. For fre0r009.a200_elaboration the computation time of R-tree structure and Hash map are similar. As we used the first 50.000 records on the entire data set, we continued to do test for that amount. Since we have not given any precision values on the inputs, we only monitor without giving any error rate.

From Figure 6.6, we can observe that both structure programs terminate equally, however, we obtained a less power consumption in R-tree for this function which is 181.1 mWatt and 195.65 mWatt for Hash. While we analyze fre0r002.d220_normalizza_tasso_001 function, we derive also precision rates and compare with Hash table. The power consumption for each

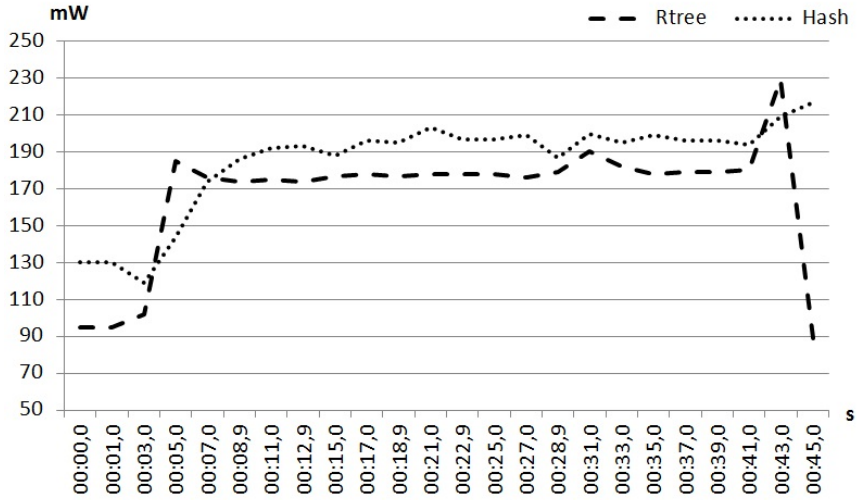


Figure 6.6: *The energy consumption and timing graph for fre0r009.a200_elaboration.*

condition is near to 170 mWatt. For Hash map, the whole energy consumption is 706.215 J, for R-tree without any precision value is 694.745 J, for R-tree giving 1% precision is 705.637 J and for R-tree giving 5% precision is 683.003 J. The results are quite similar and determining the best solution is hard after having these measurements on this function.

As a result, from the mortgage program, it is hard to decide which data structure is better since the function conditions vary. The pure mortgage functions that we use as a test set deliver us the idea of both structure works similar on timing point of view. However, on the side of test financial application, we investigate that Hash map works better then R-tree structure if precision is not under consideration.

6.3 Memoization Architecture and Trade-Off Module Validation

In this section, we validate the memoization architecture and the trade-off module. In Section 6.3.2, we tested our approach on the four functions described in Section 6.3.1.

6.3.1 Settings

We developed a tool to automatically generate casual input parameters according to a Gaussian distribution with given mean,standard deviation, and

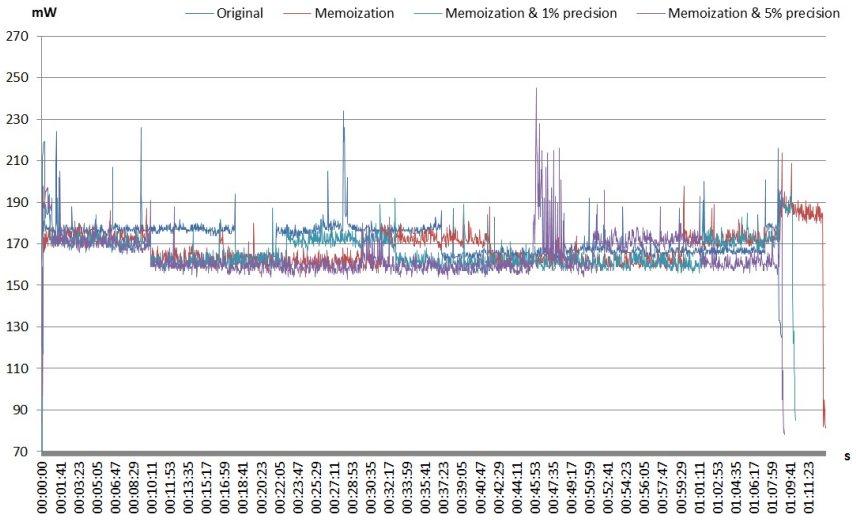


Figure 6.7: *The energy consumption and timing graph for fre0r002.d220_normalizza_-tasso_001.*

precision, determined as explained in the previous section. We executed each function a high number of times, using the input parameters generated by our tool: 100,000 for `Implied_Volatility`, `XIRR` and `Fourier`, 40,000,000 for `Black_Scholes`. Each function has been evaluated in isolation, without intervention of the memoized values replacement policy. In this case, the maximum memory assigned to JVM and memoization was set to 1GB.

Benchmark Functions Selection

Not all the pure functions detected should be memoized: some of them are not computationally intensive enough, and memoization would actually cause a slowdown if it was applied. We filter out these functions by analyzing the T_m/T_e ratio. The value of T_m (access time in memory) depends on hardware parameters of the memory hierarchy, but also on the software implementation and the size of the hash tables needed to hold the memoized values, and the Java runtime employed.

From the JavaGrande (Section 2) benchmark suite, LUFact and Crypt are discarded at this point: LUFact is a memory intensive benchmark, and Crypt has a very small memoizable function. The Fourier benchmark, on the other hand, is selected for memoization.

From the Financial set, three functions pass the selection stage, so the

set of functions that will be memoized is composed of the following:

- `Implied_Volatility`, which computes the volatility of a stock option given the market price and an option pricing model [35];
- `Black_Scholes`, which computes the price of put and call options by solving a partial differential equation according to the Black-Scholes-Merton model [10];
- `XIRR`, which computes the annualized internal rate of return of a cash flow at arbitrary points in time;
- `Fourier`, which computes the first N Fourier coefficients for function $f(x) = (x + 1)^x$.

The `Implied_Volatility` and `Black_Scholes` functions are widely used in financial applications to estimate prices and volatiliy of stock options. Similarly, the `XIRR` function is included in several banking applications to evaluate investments or credit plans. It is a computational intensive routine as it requires to find the zeros of a polynomial. The potential impact of the memoization approach on these functions is very high.

We do not have sufficient data to estimate the impact of the functions in our sample on the total consumption of a financial institution. However, it should be noted that the initial set of financial functions that we considered is limited, as we had to rely on open source publicly available code, and that our approach can be easily extended.

The chosen benchmark functions are different in size, complexity, and structure and, thus, constitute a good experimental set.

Benchmark Functions Preliminary Analysis

Before conducting our experiments, we analyzed the selected benchmark functions by applying the performance model presented in Section 4.3. Our model estimates memoization effectiveness based on time performance. According to our model the effectiveness of the memoization approach for a given function depends on the original execution time, the available memory, the precision required, and the distribution of the input parameters.

Table 6.21 reports for each function the number of considered input parameters (N_p), the estimated η for different values of the statistical mean of the combined variance of the input parameters, expressed as a percentage of the combined σ and μ values ($E\left[\frac{\sigma}{\mu}\right]$). For the sake of simplicity, we assume $\beta = 0$ in Equation 4.5 for this estimate.

Table 6.21: *Estimation of effectiveness of the memoization approach for the selected benchmarks.*

	N_p	T_e (ns)	T_m (ns)	T_m/T_e	S_d (bytes)	η at different values of $E\left[\frac{\sigma}{\mu}\right]$						
						5.00%	10.00%	15.00%	20.00%	25.00%	30.00%	35.00%
Implied_Volatility	5	2,957,629	2,000	0.07%	40	0.07%	3.07%	14.87%	27.86%	38.61%	47.02%	53.60%
Black_Scholes	5	3,581	1,600	44.69%	48	44.69%	45.74%	53.51%	64.79%	75.32%	84.08%	91.19%
XIRR	2	1,642,074	21,000	1.28%	312	1.53%	14.41%	32.73%	47.32%	55.90%	62.78%	67.92%
Fourier	3	6,442,008	1,500	0.02%	24	0.02%	1.38%	10.02%	21.75%	32.39%	41.10%	48.10%

Table 6.22: *Black Scholes: input preliminary analysis.*

Parameter	Mean	σ	τ	$N_{d,i}$	α_i	
Spot Price []	8	1	0.01	1900	0.9985	
Strike Price []	8	1	0.01	1900	0.9985	
Volatility [%]	1	0.5	0.01	400	0.9658	
Time to maturity [days]	15	15	1	359	1.0000	
Combined values (N_d, α^*, η)				518,396,000,000	0.9630	0.1145

The values of η in Table 6.21 are estimated for $S_m = 1Gb$, which corresponds to the amount of memory available for memoization that we adopted in our experiments, and for values of τ (i.e., sampling precision) compatible with real life requirements for each function. $\frac{T_m}{T_e}$ represents the maximum theoretical effectiveness, which is obtained when all the possible input values of the function are memoized ($\alpha = 1$).

Table 6.21 shows that all the functions in our sample are good candidates for memoization, as the expected savings are high even in the case of high variance of the input parameters.

To validate our approach and experimentally evaluate energy consumption savings, we identified for each function a value for the variance of input parameters that can be reasonable in real life. For example, let us consider in greater detail the `Black_Scholes` function. This function has five input parameters [10]:

- the spot price of the underlying asset;
- the strike price;
- the volatility of returns of the underlying asset;
- the risk free rate;
- the time to maturity.

Within a restricted timeframe the variation of the risk free rate is not significant. Accordingly, we assumed this input parameter fixed. In a real

world scenario, memoization tables would be recomputed when the risk free rate changes.

Table 6.22 reports reasonable parameter choices for the `Black_Scholes` function, their corresponding α_i values according to Equation 4.7. The resulting combined α^* is 0.9630, so the effectiveness η is equal to 0.1145, i.e. 11.45%, according to Equation 4.5. According to Table 6.21, we expect the average execution time of the function, when memoization is applied is approximately 45.74% of the original time.

We performed similar analysis on all the functions of our sample and we identified realistic values of σ, μ, and τ, which we used to generate workloads for our experimental campaign on energy consumption.

6.3.2 Analysis of Experimental Results of the Memoization Architecture

We tested our approach on the four functions and benchmark workloads described above, comparing it with the baseline (unmodified code). Table 6.23 reports the results of our experimental campaign. Energy consumption is computed as the integral of the additional power absorption w.r.t. the idle over the time required to process the workload. Energy consumption effectiveness, similarly to η, is computed as the ratio of the energy consumed when using our approach and the energy consumed when executing the original functions. As our approach is based on dynamic filling of memoization tables, the first times that the benchmark workload is executed most of the output values are not yet stored in the look-up table. After a transition period, the system reaches a stable state. The length of the transition period depends on the variance of the input parameters of the function. Table 6.23 reports values relative to a stable state.

Figure 6.8, 6.9, 6.10, and 6.11 show the values of the power absorbed by the whole system over time for all the experiments that we conducted, both for stable states and for transition states, when look-up tables had not been completely filled.

The results are overall encouraging. The memoization approach reduces the total energy consumption from 62.1% (`Black_Scholes`) to 99.9% (`Fourier`), with an average saving of 86%. The time performance savings range from 64.7% to 98.8%, with an average value of 87.3%. Note that time and energy savings are different by definition, as instant power consumption varies over time, but η values provide an acceptable estimate of the energy consumption effectiveness. It is also interesting to note that the effectiveness indices η estimated by means of our performance model

are fairly accurate and in all cases are conservative.

The largest savings are obtained with the `Implied_Volatility` function. This function is highly computation intensive and recursive. Moreover, it includes many computation intensive operations within a cycle, and calls another memoizable function (`Black_Scholes`). This makes the memoization approach very convenient as the same result is re-used many times by the algorithm. Generally, the more complex is the function, the greater are the savings obtained. In fact, both the `XIRR` and `Fourier` functions lead to significant savings, as they involve complex computations. Contrarily, our approach is not very convenient for the `Black_Scholes` function. This function is quite simple and requires few computational operations. The execution of the decision module introduces an overhead that translates into a higher average power (approximately +30%). This is scarcely compensated by the reduction of the execution time, enabled by the substitution of computational operations with memory accesses. As a result, the total energy consumption is reduced by 62.1%. It should also be noted that even tough the hit rate probability drops geometrically with the number of input parameters (see Equation 4.7), our approach may still bring significant savings even when applied to functions with more than one input parameter.

Table 6.23: *Execution times and energy consumption for the selected benchmarks.*

Function	Execution time as-is (s)	Execution time w/ memoization (s)	η	$\eta_{measured}$	Energy consumption as-is (J)	Energy consumption w/ memoization (J)	Energy consumption effectiveness (%)
Implied Volatility	295.76	7.48	3.07%	2.53%	7,375.03	19.39	0.26%
Black Scholes	143.22	50.41	45.74%	35.20%	4,923.23	1,865.20	37.89%
XIRR	164.21	19.60	14.41%	11.94%	4,155.84	705.50	16.98%
Fourier	644.20	7.78	1.38%	1.21%	16,655.84	16,655.84	0.06%

6.3.3 Validation of Trade-off Module

We employ the same set-up used in Section 6.3.1. In this case, we use a mix of invocations of the 4 benchmark functions, with different frequencies. We measured the execution time and energy consumption of the following scenarios:

- original code;
- memoization engine with statical allocation of memory (no trade-off module);
- memoization engine with dynamical allocation of memory driven by the trade-off module.

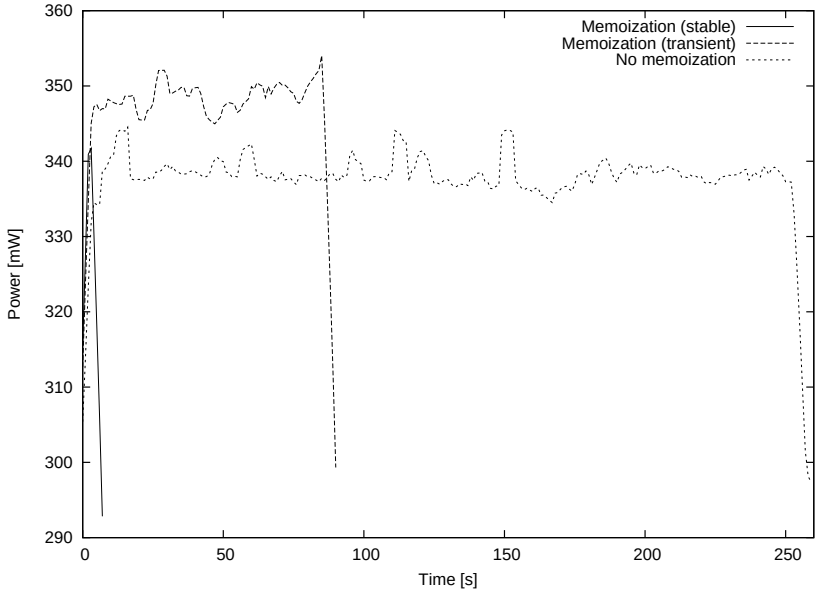


Figure 6.8: *Total power consumption for* `Implied_Volatility` *memoized function.*

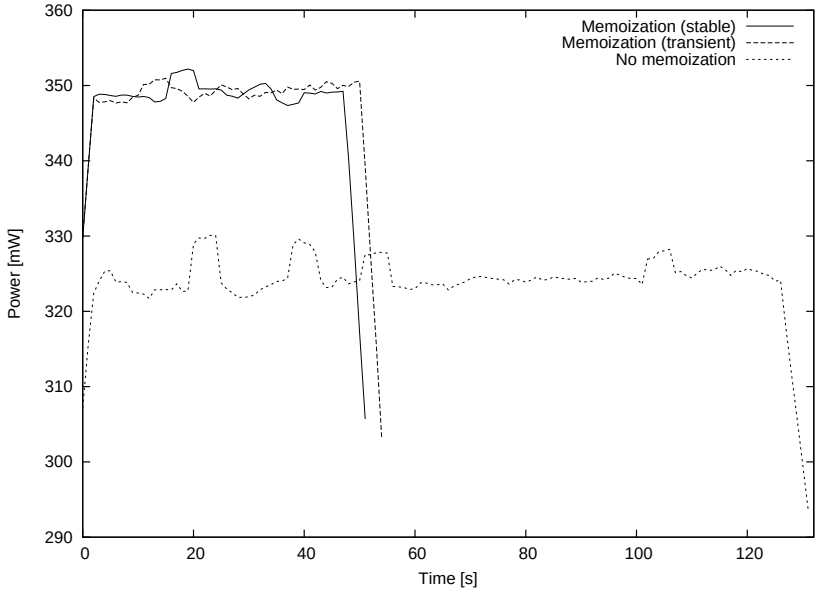


Figure 6.9: *Total power consumption for* `Black_Scholes` *memoized function.*

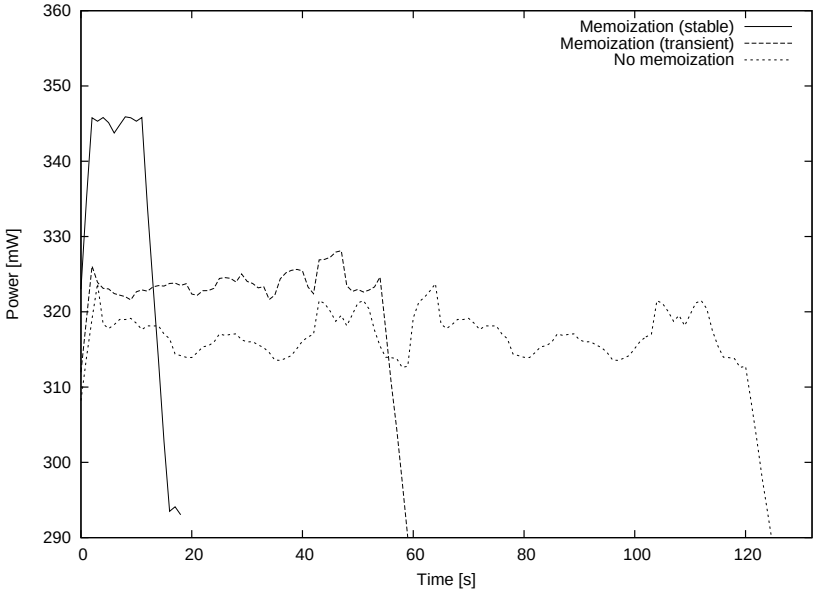


Figure 6.10: *Total power consumption for* `XIRR` *memoized function.*

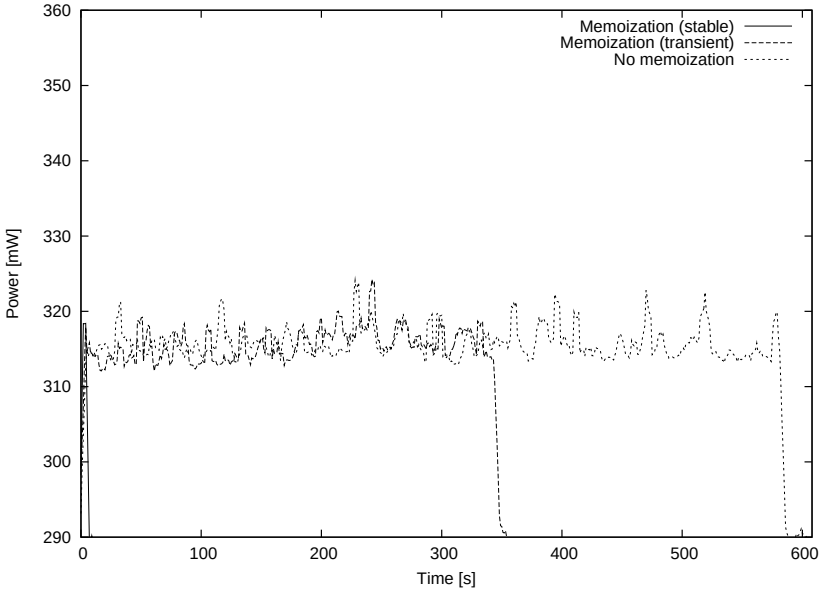


Figure 6.11: *Total power consumption for* `Fourier` *memoized function.*

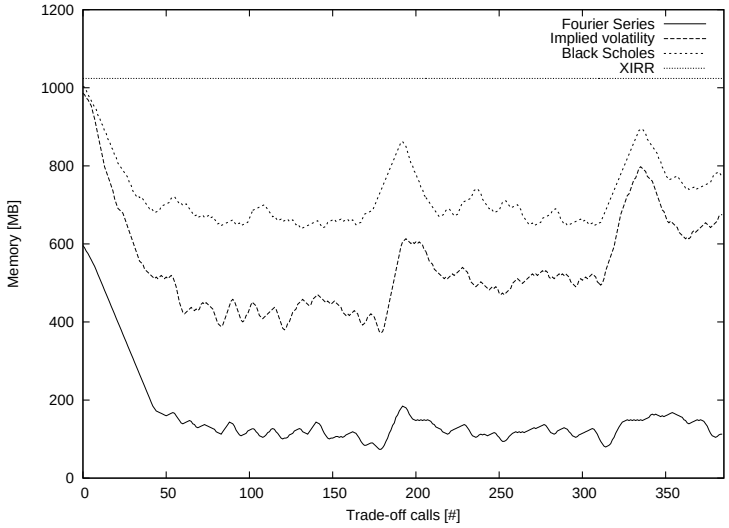


Figure 6.12: *Memory allocation distribution during the execution of the first test (40 million calls, functions invoked with equal and constant frequencies) with trade-off module active.*

We perform two different kinds of tests. First, we evaluate the impact of the trade-off module on the initial allocation of memory, keeping the function invocation frequencies constant. Second, we evaluate the behaviour of the trade-off module when the function invocation frequencies change over the time.

For the first test, we initialize memory allocation (with a total reserved memory of 1GB). The functions are invoked with an equal frequency, with a total of 40 million calls. In the scenario with memoization engine active and trade-off module disabled, calls are memoized as long as the memory allocated to each function is not full. In the third scenario, the trade-off module alters the memory allocation during the execution. Figure 6.12 reports the evolution of memory allocation in this scenario. It is evident that the trade-off module greatly helps effectively allocate memory even in a context where function invocation frequencies do not change. In fact, whereas the previsional model is based on statistical estimations of the hit rate under the hypothesis that input parameters follow a Gaussian distribution, the trade-off module considers the actual values.

For the second test, we start with the optimal memory allocation defined by the trade-off module at the end of the first test. We then change the function invocation frequencies to the following relative distributions (with a total number of invocations of 1 million):

- 25% `Fourier`, 25% `Implied_Volatility`, 25% `Black_Scholes`, 25% `XIRR`;
- 70% `Fourier`, 10% `Implied_Volatility`, 10% `Black_Scholes`, 10% `XIRR`;
- 10% `Fourier`, 70% `Implied_Volatility`, 10% `Black_Scholes`, 10% `XIRR`;
- 10% `Fourier`, 10% `Implied_Volatility`, 70% `Black_Scholes`, 10% `XIRR`;
- 10% `Fourier`, 10% `Implied_Volatility`, 10% `Black_Scholes`, 70% `XIRR`.

In the scenario with the trade-off module disabled, the allocation of the memory never changes, leading to a reduced effectiveness of the approach. On the other hand, when the trade-off module steps in, the allocation of the memory is changed dynamically to maximize effectiveness.

Table 6.24 and Table 6.25 report the energy consumption and the execution time for this test as a function of invocation frequencies relative distribution. The mean energy savings between the as-is case and case with memoization and trade-off is 96.8%, with a peak of 98.79%. Instead, the mean execution time saving is 97%, with a peak of 99%. The five invocations frequencies scenarios described above were performed a number of times sufficient to obtain measures with relative error less than 5% with confidence at 95%. Figures 6.13, 6.14, 6.15, 6.16, and 6.17 show the power consumption for the second tests for the 3 different scenarios considered. The results show that the overhead introduced by the trade-off module is largely repaid by the benefits it brings in terms of both energy consumption and execution time.

Table 6.24: *Energy consumption for the selected benchmark tests.*

Tests F_IV_BS_X (%)	Energy as-is (J)	Energy w/ memoization (J)	Savings as-is vs memo (%)	Energy w/ memoization + trade-off (J)	Savings memo vs memo+TO (%)	Total savings as-is vs memo+TO (%)
25_25_25_25	47,556.58	5,539.93	88.35%	574.97	89.62%	98.79%
70_10_10_10	141,296.21	49,921.74	64.67%	1,345.75	97.30%	98.79%
10_70_10_10	19,859.22	2,639.37	86.71%	410.80	84.44%	97.93%
10_10_70_10	25,536.81	19,827.62	22.36%	418.14	97.89%	98.36%
10_10_10_70	31,626.52	24,135.51	23.69%	3,175.51	86.84%	89.96%

Table 6.25: *Execution times for the selected benchmark tests.*

Tests F_IV_BS_X (%)	Execution time as-is (s)	Execution time w/ memoization (s)	Savings as-is vs memo (%)	Execution time w/ memoization + trade-off (s)	Savings memo vs memo+TO (%)	Savings as-is vs memo+TO (%)
25_25_25_25	1,659.63	216.74	86.94%	18.37	91.52%	98.89%
70_10_10_10	5,125.10	1,840.10	64.10%	51.30	97.21%	99.00%
10_70_10_10	781.64	103.35	86.78%	17.66	82.91%	97.74%
10_10_70_10	890.14	746.80	16.10%	15.45	97.93%	98.26%
10_10_10_70	1,130.42	933.55	17.42%	100.83	89.20%	91.08%

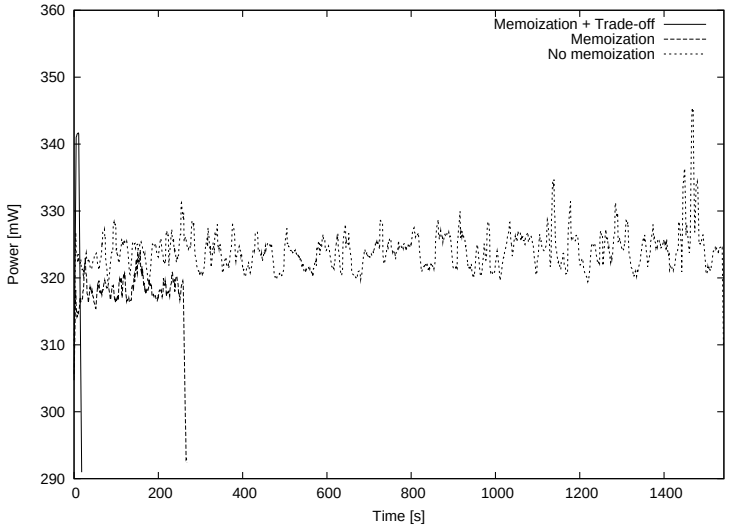


Figure 6.13: *Total power consumption for the five tests of the trade-off module: Test 25%* `Fourier`*, 25%* `Implied_Volatility`*, 25%* `Black_ Scholes`*, 25%* `XIRR`*.*

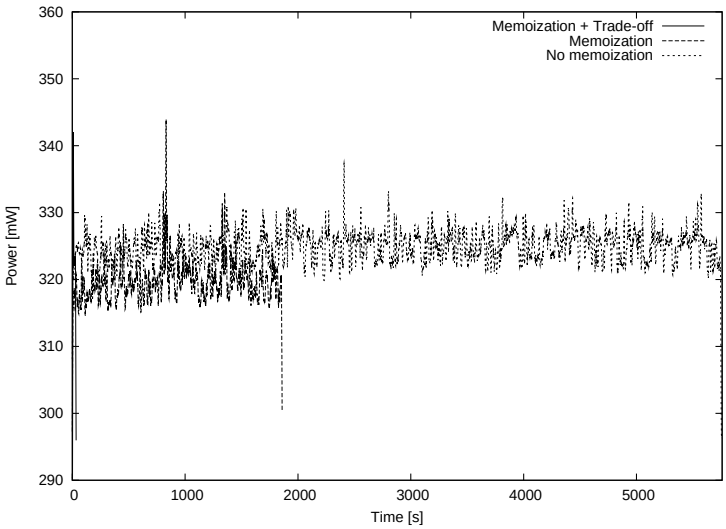


Figure 6.14: *Total power consumption for the five tests of the trade-off module: Test 70% `Fourier`, 10% `Implied_Volatility`, 10% `Black_Scholes`, 10% `XIRR`.*

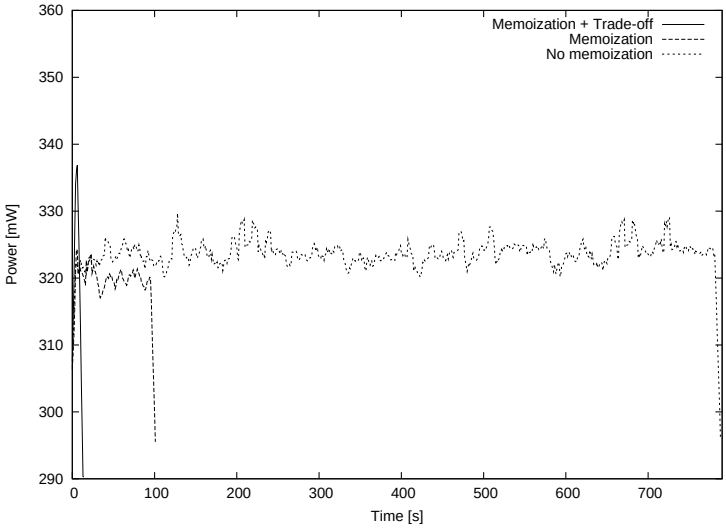


Figure 6.15: *Total power consumption for the five tests of the trade-off module: Test 10% `Fourier`, 70% `Implied_Volatility`, 10% `Black_Scholes`, 10% `XIRR`.*

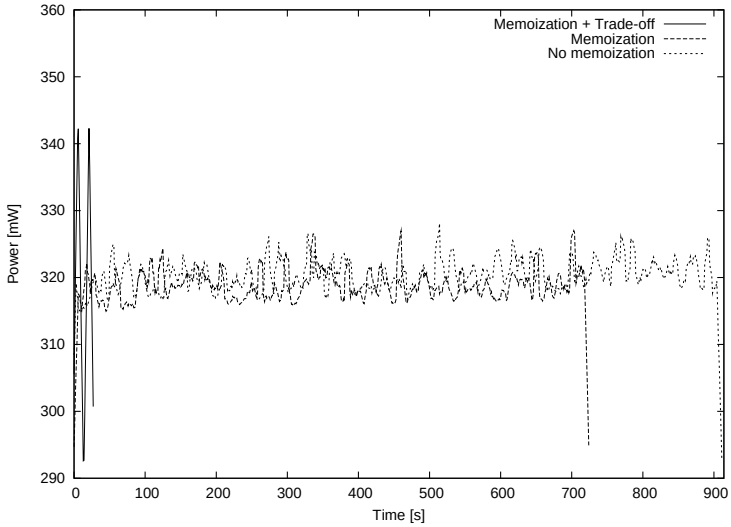


Figure 6.16: *Total power consumption for the five tests of the trade-off module: Test 10% `Fourier`, 10% `Implied_Volatility`, 70% `Black_Scholes`, 10% `XIRR`.*

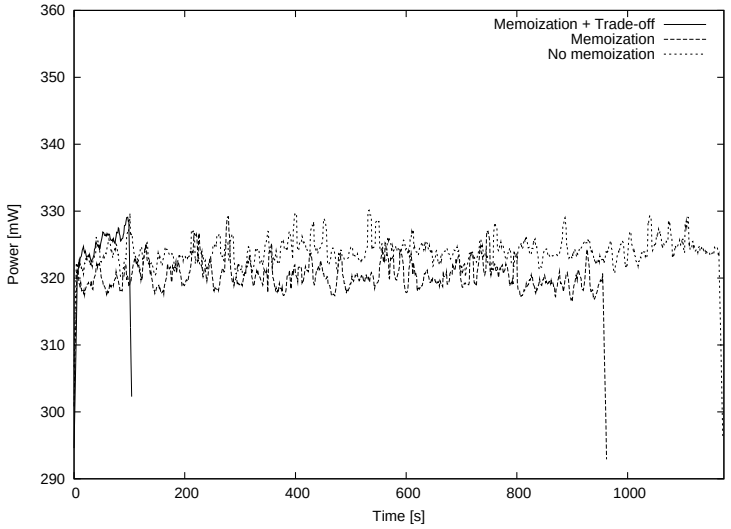


Figure 6.17: *Total power consumption for the five tests of the trade-off module: Test 10% `Fourier`, 10% `Implied_Volatility`, 10% `Black_Scholes`, 70% `XIRR`.*

CHAPTER 7

CONCLUSIONS AND FUTURE DIRECTIONS

This thesis work approaches the green IT problem from the software energy efficiency point-of-view. As illustrated in this thesis, although software does not directly consume energy, it has an impact the energy consumption of IT equipments. Consequently, software is indirectly responsible of energy consumption. Thus, there is a need for optimizing the code of application software. The literature explains how optimizing software algorithms can influence on software energy efficiency. However, there is broad evidence of the practical hurdles involved in optimizing software algorithms, tied to the technical skills of programmers, to the role of domain knowledge, and to the need for massive code refactoring and related costs. This thesis focuses on software energy consumption and provides a methodology to identify and reduce energy inefficiencies automatically.

As a first step, we have designed a methodology to estimate the energy consumption of software applications based on their usage of computing resources. This methodology has been validated on a sample of commonly used SAP transactions in a large multinational company. Our methodology can be useful to control the actual usage of resources of a data center along the energy efficiency dimension. Current accounting models are mainly based on data center floor space occupation, without considering the actual

usage of computational resources and, thus, energy consumption of a data center. We introduced three different metrics in order to decouple hardware from software energy consumption. γ provided the resource consumption related to the software application/transaction while β defined the resource usage cost. From our results, we can further assure that the γ metric is not influenced by the hardware architecture or by the OS that have been chosen for execute the applications/transactions.

The combination of γ and ω can be used to estimate the energy consumption of applications/transactions. Thus, these metrics can be used also to define classes of energy efficiency within each cluster, thus enabling the comparison of the energy efficiency of functionally similar applications/-transactions. This provides new metrics to evaluate software quality and to select applications based on their energy efficiency. It can also help to assess the impact of software customization on energy efficiency. Benchmarks are used to identify energy inefficient software applications.

The benchmarking methodology allows you to compare the energy behavior of various software applications, and then provides additional quality parameters to assess market solutions and software vendors. It may also be a valuable additional tool to evaluate both performance of the internal development groups or system integrators (e.g. related to ERP transactions customization). While for a large company ERP system change could result in very high costs, to control and optimize the operation of the system integrator may prove to be an effective lever to reduce operating costs. In fact, ERP systems customization can be the cause of significant losses in efficiency.

Our methodology can be also useful to control the actual usage of resources of a data center along the energy efficiency dimension. Current accounting models are mainly based on data center floor space occupation, without considering the actual usage of computational resources and, thus, energy consumption of a data center. Electricity may account for up to 15% of the operating costs [29] and, thus, a more accurate monitoring model may encourage significant organizational changes and allow greater efficiency. These advantages may acquire an even greater importance in a cloud environment.

It may also be of interest to apply the methodology for estimating the IT resources' consumption of a particular business process, for the treatment of specific document or transaction used by a particular user profile. In addition, the energy consumption values of different applications and transactions recorded in various business settings allow you to define the market benchmark and to compare them between different organizations

from the point-of-view of application software energy efficiency. Although there are several benchmarking initiatives related to devices energy efficiency (from electrical appliance to ICT device as computers, monitors and printers) there are not similar initiatives in the field of software application. The benefits related to these initiatives (e.g. Energy Star) are varied. Most notably the introduction of labeling of merit and the promotion of agreements from suppliers to deliver products and services more environmentally friendly . The environmental labels are a valuable tool in the hands of the consumer (or the office staff purchases in the case of businesses) in order to fairly assess the products and services that are acquired without possessing advanced technical skills and without having to make costly qualifying. Whether labeling efficiency of the application software is applied to packages sold by the software house (or to the services integration and customization offered by other market players) would allow a more responsible choice and a more comprehensive evaluation of its suppliers. The methodology presented in this thesis allows to estimate the energy consumption of applications and thus constitutes the first step towards these types of initiatives.

As a second step, we have proposed the use of memoization of Java methods to reduce energy consumption of the inefficient software applications. To this end, we have introduced an appropriate definition of weak purity and constructed a prototype that performs pure functions identification as well as automated memoization.

Our framework includes the identification of pure functions and the reengineering of non pure ones in equivalent versions, with at least a pure code section within them. The first element performs the analysis of generic functions at the bytecode level, from a twofold perspective (data dependencies and purity identification), in order to determine whether a method could be effectively considered pure. The second component, modifies the code structure of these methods, in order to obtain an equivalent version with better characteristics in terms of purity. It is able to transform a non pure function in an equivalent pure one, without impacting on the correctness of the result.

The validation of the bytecode analysis module has shown significant results. All the pure functions of the three selected benchmark suites are correctly identified, without any false negative-positive, resulting in a correctness of 100%. The analysis related to the source of impurities for the same functions, on the other hand, fails to reach such value, settling however over 99%, globally. The motivations for such discrepancy reside in the presence of particularly complex bytecode expressions, which are not cor-

rectly addressed from the analyzer. However, the corresponding extension of the implemented algorithm can meet the complete correctness requirement.

The results obtained from the bytecode modification module are very encouraging. Firstly, the correctness of the modified methods has been tested, demonstrating the accuracy of the approach in terms of bytecode manipulation. In particular, we have found that the modified modules show the same input-output behavior as the original code. Then, both the impact of code redesign and the time performance of modified modules have been evaluated. Experimental results have shown that the decrease of performance consequent code redesign is very modest (generally within a 5% threshold). For some methods, even small improvements have been obtained (up to 10% for the TSP benchmark). The reasons for these results can be found in the efficient re-engineering of code, which minimizes the use of global variables. However, not all methods flagged as modifiable have been effectively redesigned. A first selection has been performed, in order to determine the most suitable subset of functions. In fact, it has been demonstrated that some re-engineering approaches are not convenient for methods that do not perform significative computation. We also developed a trade-off algorithm that effectively allocate limited memory resources to different functions in order to maximize overall energy savings. The experimental campaign shows that, by selecting computation intensive pure functions as memoization candidates, it is possible to reap significant benefits in terms of both time and energy performance.

Our results showed that our memoization approach is useful in order to reduce both time and energy consumption of software applications. Our novelty approach is also interesting because we face the use of memoization from a different point-of-view. We analyze and implement a solution that can be applied directly to third-parts Java software applications. In fact, our approach, can act directly to the Java bytecode without the need of the source code. In this thesis is proved that our suite can analyze and modify any Java software application in order to find pure function that can be memoized. In addition these functions can be directly modified on the bytecode level introducing the bytecode instructions that perform the memoization approach. Thus, without any knowledge related to the application domain and without highly-skilled (and expensive) developers, the any software application that satisfies purity characteristics can be optimized from the energy point-of-view.

7.1 Future works

A limitation of our work is that we have considered software transactions in isolation, i.e. without saturating the computational units. Contexts characterized by highly parallel applications should be further investigated. Moreover, we have assumed that the characteristics of the hardware architecture can influence unit power absorption only, but not the flow of instructions. This is usually a good approximation in a corporate context where market standard hardware devices are deployed. However, in future work, we will further investigate the impact of more custom hardware architectures on resource usage and, consequently, on software energy efficiency.

Our software suite implementation presents some limitations, which will be addressed in future work. An additional approach that can be implemented could operate only on subsections of methods. In this case, it is necessary to isolate the set of operations that will be implemented in separate functions, along with the variables needed for the computation. Considering the dependency graph, this operation corresponds to the identification of a subset of nodes that constitutes a subgraph, in which the incoming arcs define the required variables, and the outcoming ones define results. However, to effectively implement such an approach, it is necessary to solve the problem of the enumeration of all possible subgraphs for a given graph, and consider only the good candidates for the operation; otherwise, a random choice could not only result in a worse, new version of the function, but even compromise the correctness of the execution. A last, significant improvement of the proposed work would be the implementation of a measure that numerically defines how convenient is the re-engineering of a method (for example, towards its possible memoization). The assessment should consider the impact of the bytecode modifications needed to purify the function (in terms of computation overhead) and an estimate of the function's results. In addition to that, the proposed approach could be strengthened by studying the dynamic detection of pure functions.

BIBLIOGRAPHY

[1] Java Grande suite - http://www.epcc.ed.ac.uk/research/java-grande/.

[2] Optimization package - http://www1.fpl.fs.fed.us/optimization.html.

[3] Perf4j website - http://perf4j.codehaus.org/.

[4] *Gartner Symposium/ITxpo*, 2007.

[5] *Goodwill Teams with Electronic Recyclers to Recycle eWaste*. The Free Library, 2007.

[6] G. Agosta, M. Bessi, E. Capra, and C. Francalanci. Dynamic memoization for energy efficiency in financial applications. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11. IEEE Computer Society, 2011.

[7] Giovanni Agosta, Marco Bessi, Eugenio Capra, and Chiara Francalanci. Automatic memoization for energy efficiency in financial applications. *Sustainable Computing: Informatics and Systems*, 2(2):105–115, 2012.

[8] Mir Azam, Paul Franzon, and Wentai Liu. Low power data processing by elimination of redundant computations. In *Proceedings of the 1997 international symposium on Low power electronics and design*, pages 259–264. ACM, 1997.

[9] Luiz André Barroso and Urs Holzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[10] F. Black and M. Scholes. The pricing of options and corporate liabilities. *The journal of political economy*, 81(3):637–654, 1973.

[11] Giulio Boccaletti, Markus Löffler, and Jeremy M Oppenheim. How it can cut carbon emissions. *McKinsey Quarterly*, pages 1–5, 2008.

[12] Paolo Bonzini and Laura Pozzi. Polynomial-time subgraph enumeration for automated instruction set extension. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1331–1336. EDA Consortium, 2007.

[13] Richard Brown et al. Report to congress on server and data center energy efficiency: Public law 109-431. 2008.

[14] Brendon Cahoon and Kathryn S McKinley. Data flow analysis for software prefetching linked data structures in java. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pages 280–291. IEEE, 2001.

[15] E. Capra, G. Formenti, C. Francalanci, and S. Gallazzi. The impact of mis software on it energy consumption. 2010.

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

[17] Eugenio Capra and Francesco Merlo. Green it: everything starts from the software. 2009.

[18] Alexander Chatzigeorgiou and George Stephanides. Energy metric for software systems. *Software Quality Journal*, 10(4):355–371, 2002.

[19] Norman H Cohen. Eliminating redundant recursive calls. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(3):265–299, 1983.

[20] Transaction Processing Performance Council. Tpc benchmark c: Standard specification revision 5.2, december 2003. *URL: http://www.tpc.org/tpcc/default.asp*, 2003.

[21] Markus Dahm, J van Zyl, and E Haase. The bytecode engineering library (bcel), 2003.

[22] Kaivalya M Dixit. The spec benchmarks. *Parallel Computing*, 17(10):1195–1209, 1991.

[23] Brian Duffy, Hamish Carr, and Aaron Quigley. Efficient clustering for interval trees.

[24] Christos Faloutsos and Volker Gaede. Analysis of n-dimensional quadtrees using the hausdorff fractal dimension. 1996.

[25] John Field and Tim Teitelbaum. Incremental reduction in the lambda calculus. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 307–322. ACM, 1990.

[26] Matthew Finifter, Adrian Mettler, Naveen Sastry, and David Wagner. Verifiable functional purity in java. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 161–174, 2008.

[27] William Fornaciari, Paolo Gubian, Donatella Sciuto, and Cristina Silvano. Power estimation of embedded systems: a hardware/software codesign approach. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 6(2):266–275, 1998.

[28] Roger Hoover. Alphonse: Incremental computation as a programming abstraction. In *ACM SIGPLAN Notices*, volume 27, pages 261–272. ACM, 1992.

[29] SL Josselyin, B. Dillon, M. Nakamura, R. Arora, S. Lorenz, T. Meyer, R. Maceska, and L. Fernandez. Worldwide and regional server 2006-2010 forecast. *IDC report, November*, 2006.

[30] Nagarajan Kandasamy, Sherif Abdelwahed, and John P Hayes. Self-optimization in computer systems via on-line control: Application to power management. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 54–61. IEEE, 2004.

[31] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*, pages 1–8. USENIX Association, 2010.

[32] Hua Liu, Manish Parashar, and Salim Hariri. A component-based programming model for autonomic applications. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 10–17. IEEE, 2004.

[33] Yanhong A Liu and Scott D Stoller. Dynamic programming via static incrementalization. In *Programming Languages and Systems*, pages 288–305. Springer, 1999.

[34] Yanhong A Liu, Scott D Stoller, and Tim Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(3):546–585, 1998.

[35] S. Mayhew. Implied volatility. *Financial Analysts Journal*, 51(4):8–20, 1995.

[36] Donald Michie. Memo functions and machine learning. *Nature*, 218(5136):19–22, 1968.

[37] Simon Mingay. 10 Key Elements of a Green IT Strategy. 2007.

[38] San Murugesan. Harnessing green it: Principles and practices. *IT professional*, 10:24–33, 2008.

[39] Marcio FS Oliveira, Ricardo Miotto Redin, Luigi Carro, Luıs da Cunha Lamb, and Flávio Rech Wagner. Software quality metrics and their impact on embedded software. In *Model-based Methodologies for Pervasive and Embedded Software, 2008. MOMPES 2008. 5th International Workshop on*, pages 68–77. IEEE, 2008.

[40] Steven Pelley, David Meisner, Thomas F Wenisch, and James W VanGilder. Understanding and abstracting total data center power. In *WEED 2009, Workshop on Energy-Efficient Design*, 2009.

[41] William Pugh. An improved replacement strategy for function caching. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 269–276. ACM, 1988.

[42] F Renzi. L'innovazione che fa la differenza - la strategia ibm e la tecnologia a supporto della flessibilità d'impresa e dei risparmi energetici, 2008.

[43] Hanan Samet. Deletion in two-dimensional quad trees. *Communications of the ACM*, 23(12):703–710, 1980.

[44] Alan C. Shaw. Reasoning about time in higher-level language software. *Software Engineering, IEEE Transactions on*, 15(7):875–889, 1989.

[45] Murali Sitaraman, Greg Kulczycki, Joan Krone, William F Ogden, and ALN Reddy. Performance specification of software components. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 3–10. ACM, 2001.

[46] Avinash Sodani and Gurindar S Sohi. *Dynamic instruction reuse*, volume 25. ACM, 1997.

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

[48] Witold Suryn, Alain Abran, and Alain April. Iso/iec square: The second generation of standards for software product quality. In *7th IASTED International Conference on Software Engineering and Applications*, 2003.

[49] Amin Vahdat, Alvin Lebeck, and Carla Schlatter Ellis. Every joule is precious: The case for revisiting operating system design for energy efficiency. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, pages 31–36. ACM, 2000.

[50] Elaine J. Weyuker and Filippos I. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *Software Engineering, IEEE Transactions on*, 26(12):1147–1156, 2000.

[51] Haiying Xu, Christopher J. F. Pickett, and Clark Verbrugge. Dynamic purity analysis for java programs. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 75–82. ACM, 2007.

[52] J. Zhao, I. Rogers, C. Kirkham, and I. Watson. Pure method analysis within Jikes RVM. In *Third International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS 2008),(Paphos (Cyprus))*, 2008.