Università
Cà Foscari
Venezia

Corso di Laurea magistrale *(ordinamento ex D.M. 270/2004)*
in Informatica

Tesi di Laurea

—

Cà Foscari
Dorsoduro 3246
30123 Venezia

A model for power management of cpus via frequency control

Relatore
Dott. Andrea Marin

Laureando
Simone Golfetto
Matricola 825210

Anno Accademico
2011/2012

Abstract

Dynamic voltage and frequency scaling (DVFS) is one of the most popular techniques used to reduce the power consumption of a CPU. The effectiveness of this approach is closely related to the correct detection of the workload subjected to the CPU. Often, it is difficult to detect a very fast change in load level. Moreover, the performance improvement in terms of energy consumption, is linked to a degradation of the system throughput. Currently, to the best of our knowledge, there are no analytical models that enable the computation of the energy performance of a frequency regulatory policy. This make difficult to directly evaluate the trade-offs between the system throughput and its power consumption.

In this thesis, we present a formal model for the computation of the relevant performance indexes (throughput and power consumption). The model has been developed with the ideal aim of minimizing the power consumption without affecting the throughput of the system and is formalized in Performance Evaluation Process Algebra (PEPA). This allows us to efficiently obtain the required performance indexes in a purely algorithmic way, by the analysis of the underlying Markov chain. Then, the model has been applied to evaluate the performance of a set of frequency control strategies on the traces of the *GoogleClusterData* project, that provides detailed information about the workload of a cluster of about 11,000 machines. In order to parameterize the model, the workload traces have been fitted into stationary Markovian processes. The performance of the model has been evaluated using the ECLIPSE plugin provided by PEPA group.

Extensive testing highlights how the proposed model provides important opportunities to reduce the power consumption of CPUs while maintaining reasonable level of throughput. Possible applications of the proposed framework include the reduction of power consumption in large data centers but also the definition of smart management policies of battery in mobile devices such as tablets and smartphones.

Sommario

Il Dynamic voltage and frequency scaling (DVFS) è sicuramente una delle tecniche più conosciute ed utilizzate per tentare di ridurre il consumo energetico di una CPU. L'efficacia di questo approccio è tuttavia, strettamente legata alla corretta rilevazione del carico di lavoro sottoposto alla CPU. Spesso infatti, risulta difficile rilevare un cambiamento molto veloce nel livello di carico. Inoltre, il miglioramento delle prestazioni in termini di consumo energetico, è ostacolato dalla degradazione del livello di throughput del sistema. Attualmente, non si è a conoscenza dell'esistenza di modelli analitici che permettano di calcolare il rendimento energetico delle politiche di controllo della frequenza. Ciò rende difficile valutare il trade-off tra il throughput del sistema ed il consumo di energia in maniera rapida.

In questa tesi, viene presentato un modello formale per il calcolo degli indici di performance (throughput e consumo di energia). Il modello è stato sviluppato con lo scopo ideale di ridurre al minimo il consumo energetico senza però compromettere la produttività del sistema ed è formalizzato nell'algebra di processo PEPA. Ciò permette, attraverso l'analisi della catena di Markov sottostante il modello, di ricavare gli indici richiesti in maniera puramente algoritmica, oltre che efficace. Il modello è stato poi testato al fine valutare le prestazioni di un insieme di strategie per il controllo dinamico del livello di frequenza, sulle tracce messe a disposizione dal progetto *GoogleClusterData*, che fornisce informazioni dettagliate sul carico di lavoro di un cluster di circa 11.000 macchine. La parametrizzazione del modello ha riguardato la trasformazione delle tracce in processi Markoviani e le prestazioni sono state valutate utilizzando il plugin per ECLIPSE fornito dal team PEPA.

I test eseguiti evidenziano come il modello proposto offra importanti opportunità nella riduzione del consumo energetico delle CPU mantenendo, allo stesso tempo, un ragionevole livello di throughput. Le possibili applicazioni del framework proposto comprendono la riduzione del consumo energetico nei grandi centri dati, ma anche la definizione di nuove politiche per la gestione intelligente della batteria in dispositivi mobili come tablet e smartphone.

Contents

List of Figures

List of Tables

Listings

Preface

In this thesis we present a framework to evaluate the performance of a frequency regulatory policy in terms of different kinds of metrics, such as throughput and power loss. It consists of two parts. The first part introduces the formalisms used in the following. The second part illustrates the contributions of this work. Chapter 1 introduces the Markov processes. The aim is to give a brief overview, presenting their characteristics and common properties. We focus, in particular, on the study of the basic theory of Markovian stochastic processes with a discrete state space and continuous time. Chapter 2 briefly illustrates the Performance Evaluation Process Algebra (PEPA), a stochastic process algebra that extends the classical algebras by introducing a rate for the transitions and probabilistic branching. Particular attention is given to syntax, semantics and to the underlying stochastic process. We also introduce a method to derive the performance measures.

The second part of the thesis illustrates the original contributions. Chapter 3 clarifies how the model is translated in PEPA language. We also explain the computation of the performance indexes through the structures provided by the process algebra. Chapter 4 introduces the algorithm for the automatic formalization in process algebra of the model and for the measures collection. We present an accurate description of the developed procedures and their parametrization, explaining the set of regulatory policies and the continuos-time Markov chains. Then the results are presented. Chapter 5 shows the behavior of the model when it is parameterized starting from a real CPU workload trace. In order to parameterize the model, we translate the traces of a *Google* cluster into Markovian process and evaluate the performance measures.

For the sake of readability, Appendix A provides the code developed to automatize the performance index computation procedure. Appendix B shows the code to extract the workload of a specific machine of the Google cluster. Finally we present the conclusions of the work.

Introduction

In recent years, the energy efficiency issue of the electronic devices has played an important role. From the smallest mobile device to the largest data center, the race to achieve the best energy performances had led the manufacturers to face with new challenges. In fact, while the increasing demand for computational resources leads to the development of systems whose energy requirement becomes higher and higher [10], the increasing of energy costs and a renewed environmental awareness, require to address the issue in a decisive manner [20]. However, when we talk about energy consumption and energy efficiency, usually we tend to underestimate the problem. The reduction in the energy consumption is a challenge that involves a big set of heterogeneous devices, from smartphones and tablets to personal computers and from servers to the larger data centers.

Modern mobile devices are capable to perform tasks that until a few years ago were unthinkable, reaching a level which can be compared to personal computers. A last generation tablet or smartphone is able, in fact, to access the Internet, process audio and video media, serve as a navigation system up to be a real portable game console. All operations that require a considerable computational power to maintain an adequate user experience. These devices are by definition related to energy consumption since their energy comes from a battery. Power consumption and battery life is one of the most central issues in mobile platforms. This problem has been widely investigated [6] and many proposals on new energy management have been made [28, 9].

On the other hand, the energy efficiency issue manifests itself with an even more powerful, talking about the data centers. In fact, the problem within reach even higher levels. In large data centers, energy consumption comes not only from the consumption of the individual elements that make it up but also all infrastructure that must maintain its operations. It passes from refrigeration equipment that must dispose of the large amount of heat produced by the machines, to the UPS that needs to be offset to a possible lack of electricity [20]. It is understood, therefore, that in this case the search for better solutions involves numerous fields of study. There are several strategies and techniques that over the years have been researched and tested for this purpose. However, what appears to be the starting point to try to obtain the best energy performance is definitely the element that performs the computation, namely the CPU.

Dynamic voltage and frequency scaling (DVFS) is one of the most popular techniques used to reduce the power consumption of a CPU. Proposed for the first time by Weiser [27], DVFS is a popular technique used to save power on a wide range of devices. It is able to

reduce the power consumption of a CMOS integrated circuit by reducing the frequency at which it operates. Many DVFS schemes have been widely explored and applied in different areas [2, 8]. In general in the energy management it is important to grant an adequate degree of system response time and throughput. Many techniques are designed to shut down parts of the system in order to conserve energy for resulting in a strong performance degradation. Then, the goal is to design efficient algorithms that are able to offer a good tradeoff between performance and power consumption.

The problem of assessing the energy performance through an algorithmic methodology, it is becoming increasingly important. In [11], for example, the authors present a method for the automatic evaluation of performance in Mobile Ad-hoc Networks (MANETs). In this thesis we propose a framework to automatically evaluate the performance of a frequency regulatory policy in terms of different kinds of metrics, such as throughput and power consumption. The model has been developed with the ideal aim of minimizing the power consumption without affecting the throughput of the system and is formalized in Performance Evaluation Process Algebra (PEPA) [12]. This allows us to efficiently obtain the required performance indexes in a purely algorithmic way by the analysis of the underlying Markov chain.

The heart of the thesis consists of three fundamental parts. The first part is called *Model defintion*. In this stage we will give the definition of the developed model, exploring in detail the construction process. What we will do is formalize through the process algebra PEPA, two distinct models. The first model, represents the different workload levels of a generic CPU. Each level is expressed in terms of operative frequency. This model will be constituted by a completely connected graph in which, each state, represents a workload level. To make the modeling process easier, our analysis is restricted to only four different levels. The second model determines the frequency regulatory policy of the CPU. A regulatory policy is typically constituted of two elements: a set of levels representing the clock speeds of CPU and an algorithm that manages the frequency switch. Assuming to model a single core CPU, we will formalize a model in which the states constitute the available frequency level and the transitions between them, represent the chances to change level. Also in the frequency model, each level is expressed in terms of operative frequency. In order to simplify the whole process, we will consider only policies with two and three frequency levels. The cooperation between the two models will be guaranteed by PEPA. Through the concepts of shared actions and passive rate, both provided by the process algebra, it will be possible to model a CPU which changes its operating speed (frequency level) according with the workload level detected at different time instants. In this part, as last thing, we will define the performance indexes we will use to evaluate the success of the modeling process and the behavior of the frequency regulatory policies. In particular, we will focus on the measurement of system throughput and power consumption. Both measures, will be obtained through the definition of appropriate reward structures [15].

Once the formalization stage will be completed, we will analyze several aspects of the model. This part is the *Model validation*. What we'll do is parameterize the model and test its behavior through a specifically developed algorithm. The parameterization

process constitutes a fundamental parts of the modeling process. It will involve three different aspects:

1. We will develop six different frequency regulatory policies, relaying on two and three frequency levels. Each policy will determine a different balancing of the frequency switch process.

2. We will define a set of different workload level detection speeds. The detection speed is a crucial factor to correctly detect this level. A too slow speed does not provide sufficient accuracy but a speed too high leads a too much frequent frequency switch. Therefore, we will evaluate the model at different detection speeds.

3. We will parametrize the workload model. To do this, we will generate two different continuous-time Markov chains [25, 17, 5] which represent two different workload situations.

Immediately after, we will present the MATLAB algorithm specifically developed to compute the desired performance indexes. This algorithm, together with the PEPA Eclipse plugin [26], will allow to automate many of the operations required to collect and analyze the data.

The third and final part will require more effort. It consists of a possible application of the model on real data. From the data made available by the *GoogleCluster* project, we will parametrize the model. The traces of the workload of a Google's cluster will be fitted into a continuous-time Markov chain and the energy performance of the frequency regulatory policies previously defined, will be analyzed.

Part I

Formalisms and state of the art

Chapter 1

Stochastic processes

1.1 Introduction

In performance evaluation of computer systems, we often have to deal with problems whose solutions are related to random processes that evolve over time. In this kind of problems, the evolution of the processes is not totally predictable because of an intrinsic property of the system (e.g. the Internet workload) or because, the modeler, abstract out some details (e.g. the lifetime of a system component). A basic methodology to address them consists in the use of Markov processes. The Markov processes constitute a class of stochastic processes whose memoryless assumptions make them mathematically tractable. Our aim is to give a brief overview of Markov processes, presenting their characteristics and common properties. We mainly focus on the study of the basic theory of Markovian stochastic processes with a discrete state space and continuous time.

The chapter is structured as follows: in Section 1.2 we will give the definition of Markov process and Markov property and in Section 1.3 we will introduce the continuous-time Markov chains. Particular attention is reserved to the definition of the transition probability matrix and the transition rate matrix (1.3.1), to the *Chapman-Kolmogorov equations* (1.3.2) and to the transient and steady-state probability distribution (1.3.3).

1.2 Markov processes and Markov property

Formally, a stochastic process [25] is a family of random variables $\{X_t : t \in \mathcal{I}\}$ where each X_t is a random variable defined on a probability space. The parameter t represents the time, so the value assumed by the variable X at time t is denoted by X_t and is called *state*. The set $\mathcal{I} \subseteq (-\infty, +\infty)$ is called *index set* and can be either discrete or continuous. If it is discrete, e.g. $\mathcal{I} = \{0, 1, 2, \dots\}$ then the process is called *discrete-time Markov process*. If $\mathcal{I}$ is continuous, e.g. $\mathcal{I} = \{t : 0 \leq t < +\infty\}$ the process is called *continuous-time Markov process*. Even set $\mathcal{S}$ of all possible states, said *states space*, can be either discrete or continuous. If it is discrete then the process is referred to as a *discrete-space* or *chain* and the states are typically labelled with the set of natural numbers

$\mathcal{S} = \{0, 1, 2, \dots\}$. Otherwise the process is called *continuous-space*. Thus, a stochastic process may evolve at a discrete or continuous time points and can have a discrete or continuos states space. When the evolution of the process is independent of the elapsed time, the process is said *time-homogeneous*. A stochastic process constitutes a Markov chain [5] if its conditional joint distribution function (CDF) satisfies the Markov property. Formally, given the stochastic process $\{X_t : t \in \mathcal{I}\}$, it constitutes a Markov process if $\forall\, x_i \in \mathcal{S}$ and for any sequence $t_0 < t_1 < \dots < t_n < t_{n+1}$ the CDF of $X_{t_{n+1}}$ depends only on the last previous value X_{t_n} and not on the earlier values:

$$Pr\{X_{t_{n+1}} \leq x_{n+1} \,|\, X_{t_n} = x_n,\, X_{t_{n-1}} = x_{n-1}, \dots,\, X_{t_0} = x_0\} \\ = Pr\{X_{t_{n+1}} \leq x_{n+1} \,|\, X_{t_n} = x_n\} \quad (1.1)$$

Equation (1.1) is the well-known *memoryless* or *Markov property*. We can informally say that a (time-homogeneous) Markov process is a particular stochastic process in whose the state in which the system find itself at time step t_{n+1} depends only on where it is at time step t_n. Below we consider in particular the discrete-space Markov process in continuous time with a time independent pattern behavior.

1.3 Continuous-Time Markov chains

As explained in the previous section, in a continuous-time Markov chains (CTMC), the time variable associated with the system evolution is continuous. This means that the transition from a state to another state may occur in any point of time. To formally define a CTMC, we refer back to the description of Markov process given by the equation (1.1) and we specialize it [5, 17], considering a continuous index set $\mathcal{I}$ and a discrete states space $\mathcal{S}$. A stochastic process $\{X_t : t \in \mathcal{I}\}$, constitutes a CTMC if $\forall\, x_i \in \mathcal{S} = \mathbb{N}_0$, $\forall\, t_i \in \mathbb{R}_0$, $\forall\, n \in \mathbb{N}$ and for any sequence $t_0 < t_1 < \dots < t_n \; < t_{n+1}$, the following relation, that defines the Markov property of CTMC, holds:

$$Pr\{X_{t_{n+1}} = x_{n+1} \,|\, X_{t_n} = x_n,\, X_{t_{n-1}} = x_{n-1}, \dots,\, X_{t_0} = x_0\} \\ = Pr\{X_{t_{n+1}} = x_{n+1} \,|\, X_{t_n} = x_n\} \quad (1.2)$$

In same way, we can say that given the states $i, j, k \in \mathbb{N}_0$ and given the time epochs u, v, w with $u, v \geq 0$ and $0 \leq w \leq u$, the stochastic process $\{X_t : t \in \mathcal{I}\}$ is a CTMC if:

$$Pr\{X_{u+v} = k \,|\, X_u = j,\, X_w = i\} = Pr\{X_{u+v} = k \,|\, X_u = j\} \quad (1.3)$$

From the right-hand side of the equation (1.3), we can consider the *transition probability* from state i to state j in time period $[u, v)$ as $p_{ij}(u, v) = Pr\{X_v = j \mid X_u = i\}$, if the CTMC is non time-homogeneous with $u, v \in \mathcal{I}$, $u \leq v$. If the transition probability $p_{ij}(u, v)$ is independent of the actual values of u and v but depends only on the interval $\tau = v - u$, then the CTMC is time-homogeneous and we simplify the notation by writing:

$$p_{ij}(\tau) = p_{ij}(u, \tau) = Pr\{X_{u+\tau} = j \mid X_u = i\} \qquad \forall\, u \geq 0 \tag{1.4}$$

Equation (1.4) represents the probability of being in state j after time interval τ given the actual state i. Note that, in this way, the probability does not depend on the current time u, follows that:

$$\sum_{\forall\, j} p_{ij} = 1 \qquad \forall\, \tau \in \mathbb{R}^+ \tag{1.5}$$

Finally, the transition probabilities $p_{ij}(u, v)$ are usually summarized [16] in a stochastic, non-negative matrix $\mathbf{P}(u, v)$ formed by placing $p_{ij}(u, v)$ in row i and column j and in any time interval $[u, v)$.

1.3.1 Transition rates matrix

We have just seen how to represent the interactions in a CTMC by specifying a transition probability matrix $\mathbf{P}(u, v)$. To simplify the notation and the analysis, it is usually [25, 5] convenient to represent the chain in terms of the rates at which transition occurs.

Looking at $\mathbf{P}(u, v)$, is easy to see that the probability to observe a transition from a given state i depends [25, 17] on the length of observation window δu. Let $p_{ij}(u, u + \delta u)$ be the probability to observe a transition form state i to state j in the period $[u, u + \delta u)$, when $\delta u \to 0$ the elements $p_{ij}(u, u + \delta u) \to 0$ for $i \neq j$. Elements $p_{ii}(u, u + \delta u) \to 1$ for the law of the conservation of probability. Let us focus now our attention on the rates of the transitions. A rate does not depend on the size of the observation window δu. It is a quantity that denotes the number of transitions observed per unit time. Let $q_{ij}(u)$ be the rate at which transitions occurs from i to j with $i, j \in \mathcal{S}$, when the CTMC is time-homogeneous and $i \neq j$ we have:

$$q_{ij} = \lim_{\delta u \to 0} \left(\frac{p_{ij}(\delta u)}{\delta u} \right) \qquad q_{ii} = \lim_{\delta u \to 0} \left(\frac{p_{ii}(\delta u) - 1}{\delta u} \right) \tag{1.6}$$

A CTMC is therefore represented by a matrix $\mathbf{Q}(u) = [q_{ij}]$ called the *infinitesimal generator*

or *transition rate matrix*. Each element q_{ij} is the rate at which the chain moves form the state i at time u to the state j. The diagonal elements q_{ii} are defined by $q_{ii} = -\sum_{i \neq j} q_{ij}$. It is clear that the rows of the matrix sum to zero. Elements q_{ij} and q_{ii} of (1.6) can be expressed in a more elegant matrix form as:

$$\mathbf{Q}(u) = \lim_{\delta u \to 0} \left(\frac{\mathbf{P}(u, u + \delta u) - \mathbf{I}}{\delta u} \right) \tag{1.7}$$

where $\mathbf{P}(u, u + \delta u)$ is the already defined transition probability matrix and $\mathbf{I}$ is the identity matrix. When the CTMC is time-homogeneous, rates q_{ij} are independent of time and the matrix can be simply written as $\mathbf{Q}$.

1.3.2 The Chapman-Kolmogorov equations

A well-known procedure to compute the transition probabilities of a CTMC is the system of *Chapman-Kolmogorov equations*. It permits to derive the transition probabilities by splitting them [5] into two different subtransitions, passing through an intermediate state in a some intermediate time:

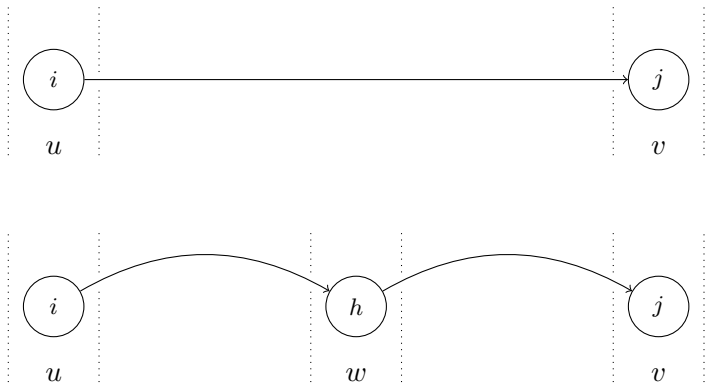

Figure 1.1: Example of transition splitting: from state i at time u to state h at time w and from state h at time w to state j at time v with $0 \leq u \leq w \leq v$.

For a non time-homogeneous CTMC the equations may be derived from the Markov property (1.2) by applying the theorem of total probability [25, 5] :

$$p_{ij}(u, v) = \sum_{\forall h \in \mathcal{S}} p_{ih}(u, w) \cdot p_{hj}(w, v) \qquad 0 \leq u \leq w \leq v \tag{1.8}$$

In the time-homogeneous case, the equation (1.8) may be rewritten as:

$$p_{ij}(u+\delta u) = \sum_{\forall\, h \in \mathcal{S}} p_{ih}(u) \cdot p_{hj}(\delta u) \qquad \delta u \geq 0 \tag{1.9}$$

Equation (1.9) cannot be solved easily and used to compute the state probabilities, but it has to be transformed into a system of differential equations. We write:

$$p_{ij}(u+\delta u) = \sum_{h \neq j} p_{ih}(u) \cdot p_{hj}(\delta u) + p_{ij}(u) \cdot p_{jj}(\delta u) \tag{1.10}$$

Dividing both sides by δt, taking $\lim_{\delta t \to 0}$ and recalling the equations (1.6), we derive the *Kolmogorov's forward equation*:

$$\frac{d\, p_{ij}(u)}{d\, u} = \sum_{\forall\, h \in \mathcal{S}} p_{ih}(u) \cdot q_{hj} \tag{1.11}$$

Altenatively we can derive and use the *Kolmogorov's backward equation*:

$$\frac{d\, p_{ij}(u)}{d\, u} = \sum_{\forall\, h \in \mathcal{S}} p_{hj}(u) \cdot q_{ih} \tag{1.12}$$

Both forward equation (1.11) and backward equation (1.12) may be expressed in matrix form as follow:

$$\frac{d\, p_{ij}(u)}{d\, u} = \mathbf{P}(u) \cdot \mathbf{Q} \qquad\qquad \frac{d\, p_{ij}(u)}{d\, u} = \mathbf{Q} \cdot \mathbf{P}(u)$$

Recalling the (1.11), we see that it, at least formally, can be solved as:

$$\mathbf{P}(u) = c \cdot e^{u\mathbf{Q}} = \mathbf{P}(0) \cdot e^{u\mathbf{Q}}$$

where $\mathbf{P}(0) = \mathbf{I}$ is the identity matrix and $e^{u\mathbf{Q}}$ is the matrix exponential. The solution is obtained [25] by putting together the last two equation, thus:

$$\mathbf{P}(u) = \mathbf{P}(0) \cdot e^{u\mathbf{Q}} = \left(\mathbf{I} + \sum_{n=1}^{\infty} \frac{u^n \cdot \mathbf{Q}^n}{n!} \right) \tag{1.13}$$

This solution is valid in the case that the states space $\mathcal{S}$ is finite. We can point out that

given a state space $\mathcal{S}$, the infinitesimal generator $\mathbf{Q}$, completely determines the Markov chain. Thus [18], it is sufficient to characterize a chain by simply providing a states space $\mathcal{S}$ and infinitesimal generator $\mathbf{Q}$.

1.3.3 Probability distributions

We focus now on the short-term or *transient* behavior of a CTMC. Let $\boldsymbol{\pi}_i(u) = Pr\{X_t = i\}$ the ith element of the probability vector $\boldsymbol{\pi}(u)$ that denotes the probability the system be in state i at time u. It is determined [25] by:

$$\boldsymbol{\pi}_i(u + \delta u) = \boldsymbol{\pi}_i(u) \cdot \left(1 - \sum_{\forall j \neq i} q_{ij} \cdot \delta u\right) + \left(\sum_{\forall h \neq j} q_{hi} \cdot \boldsymbol{\pi}_h(u)\right) \cdot \delta u + o(\delta u)$$

where the "*little oh*" operator $o(\delta u)$ is a quantity for which:

$$\lim_{\delta u \to 0} \frac{o(\delta u)}{\delta u} = 0$$

Remembering that $q_{ii} = -\sum_{\forall j \neq i} q_{ij}$, we can, as usual, express in matrix form (and time-homogeneous case):

$$\frac{d\,\boldsymbol{\pi}(u)}{d\,u} = \boldsymbol{\pi}(u) \cdot \mathbf{Q} \tag{1.14}$$

and concluding by deriving the transient solution $\boldsymbol{\pi}(u)$:

$$\boldsymbol{\pi}(u) = \boldsymbol{\pi}(0) \cdot e^{u\mathbf{Q}} = \boldsymbol{\pi}(0) \cdot \left(\mathbf{I} + \sum_{n=1}^{\infty} \frac{u^n \cdot \mathbf{Q}^n}{n!}\right)$$

Let us now turn our attention to the long-term or *steady-state* behavior of a CTMC. We have just seen that, in a time-homogeneous CTMC, the state probabilities $\boldsymbol{\pi}_i(u)$ are governed by the system of differential equations (1.14).

Informally, we can think, that the equilibrium situation has been reached when [25] the rate of change of the probability distribution vector is equal to zero. According to this the left-hand side of equation (1.14) is equal to zero and it becomes:

$$\boldsymbol{\pi}(u) \cdot \mathbf{Q} = 0 \tag{1.15}$$

The system has therefore reached a limiting distribution and the probability vector $\boldsymbol{\pi}$ no longer depends on time u. If the limit $\boldsymbol{\pi}_i = lim_{u\to\infty}\boldsymbol{\pi}_i(u)$ exists, if it is independent of the initial probability vector $\boldsymbol{\pi}_i(0)$ and if all of its components are strictly positive, then it is unique and it is called the *steady-state probability distribution*. The steady-state probability distribution may be simply obtained by solving the system of linear equations:

$$\boldsymbol{\pi} \cdot \boldsymbol{Q} = 0$$

subject to the *normalization* condition $\|\boldsymbol{\pi}\|_1 = 1$. These equations are called the *global balance equations*.

1.4 Conclusion

A Markov chain is a stochastic process in whose the state in which the system find itself at time step $n+1$ depends only on where it is at time step n and not from how it has come to this state. This is known as Markov property from the name of Russian mathematician Andrei Andreyevich Markov who first developed the theory and is the most important feature that makes Markov chains remarkably interesting. Markov chains provides very powerful, flexible and efficient means for the description and analysis of dynamic system properties.

In this chapter we have presented Markov chains in continuous time, analyzing their main features and the stationary behavior. We also introduced some basic formalisms and definitions that will be useful in the development of the thesis.

Chapter 2

Performance Evaluation Process Algebra

2.1 Introduction

Process algebras are a family of mathematical theories that provides the possibility to give a formal representation of concurrent systems. In particular, they offer useful tools to represent the system, to reason on its behavior and focus on interactions, communications, and synchronizations between a collection of independent agents or processes. Famous examples of process algebras include the *Calculus of Communicating Systems* (CCS) [19], the *Communicating Sequential Process* (CSP) [14] and the *Algebra of Communicating Process* (ACP) [4].

In this chapter we focus on the *Performance Evaluation Process Algebra* (PEPA) [12], a stochastic process algebra that extends the classical algebras by introducing a rate for the transitions and probabilistic branching. The choice of PEPA is mainly related to its strict connection with the stochastic processes. The PEPA operational semantic permits to generate a continuous-time Markov process for any PEPA model. In particular, it is shown that, under appropriate assumptions, the underlying stochastic process of the model is a continuos-time Markov chain. The CTMCs are relatively easy to analyze due to the memoryless (or Markov) property described in section 1.2. Thus any model may be simply examined and the performance indexes may be extracted.

The chapter is structured in two section. Section 2.2 defines the Performance Evaluation Process Algebra (PEPA), explaining its syntax (2.2.1) and introducing some important aspects (2.2.2). Section 2.3 show how it is possible to derive an underlying continuous-time Markov chain for each PEPA model, focusing the attention on the construction of the chain (2.3.1), on the tensorial representation of the composed models (2.3.2) and finally on the method to derive some interesting performance indexes (2.3.3).

2.2 Performance Evaluation Process Algebra

The Performance Evaluation Process Algebra (PEPA) is a stochastic process algebra designed for the first time by Jane Hillston [12] to study how the compositional features of process algebras could contribute to the practice of performance modeling.

A PEPA system consist of a countable set $\mathcal{C}$ of interacting components that may perform single or multiple activities. A component can be thought as an active part of the system, which defines the behavior of it. Furthermore, they may be atomic or may be composed of different components. The behavior of a component is defined by the activities it can engage. The term *activity* is used to underline that the engaged action is not an instantaneous action, differently from the usual process algebra notation. This is because, in PEPA, every activity has a duration determined by an exponentially distributed random variable. The parameter of this variable is called the *activity rate* and may be a positive real number or the symbol $\top$, which represent an unspecified activity rate. Each activity in the system is associated to one and only one type in the countable set $\mathcal{A}$ of all possible types. This is called *action type*. If the system is carrying out an unknown or uninteresting action the action type is set to τ (unknown action type).

An activity is thus defined as a pair (α, r) where $\alpha \in \mathcal{A}$, is the action type and r is the action rate. When the activity $a = (\alpha, r)$ is enabled, a delay is computed according to its distribution function F_A. This means that the probability that the activity a happens within a period of time of length t is:

$$F_a(t) = 1 - e^{-rt}$$

where the computed delay depends on the rate r of the activity a. This can be thought as the start of a timer made by the activity: it becomes active when the timer expires.

2.2.1 Syntax and semantics

Components and activities are the language primitives. In addition, a set of combinators is given. They allow to link together different parts of the language. The combinators enable expressions to be defined making explicit the behavior of a component. The syntax of PEPA is formally specified using the following grammar:

$$P \ ::= \ (\alpha, r).P \ \mid \ P + Q \ \mid \ P \underset{L}{\bowtie} Q \ \mid \ P \setminus L \ \mid \ A$$

where P and Q are two components, α is an action type, r is the action rate, L is a subset of $\mathcal{A}$ and A is a constant. Let us see them more in detail:

Action prefixing

The first production of the grammar is constituted by the *action prefixing* operator. It represents the most basic mechanism to define the behaviors of the components. A component $(\alpha, r).E$ perform an action type α whose duration is exponentially distributed with parameter r and then behaves as a component E. In this way the time δt, spent from the activity is determined by the probability distribution. The behavior of the action prefixing operator is formalized in structural operation semantic (SOS) by the following inference rule:

$$[\text{ACT}]\ \frac{}{(\alpha, r).E \xrightarrow{(\alpha,r)} E}$$

Competitive choice

In some cases, we have to deal with a system that may behave not only in a unique definite way, but in many different ways. The *competitive choice* operator is able to represent this kind of situations. In particular, given a component $E+F$, it enable the system to behave either as a component E or as F. This behavior is possible only if both of components E and F are able to perform an activity with the common action type. We defined this operator "competitive" because the two components are implicitly competing for the same resource. The first of them that complete the activity is allowed use the resource, the other is discarded. This behavior is well explained by the two SOS rules associated with the operator:

$$[\text{SUM}_1]\ \frac{E \xrightarrow{(\alpha,r)} E'}{E + F \xrightarrow{(\alpha,r)} E'} \qquad [\text{SUM}_2]\ \frac{F \xrightarrow{(\alpha,r)} F'}{E + F \xrightarrow{(\alpha,r)} F'}$$

It is important to note that the probability that two components complete an activity at the same time is equal to zero according to the continuous nature of the probability distribution.

Cooperation

The *cooperation* operator $E \underset{L}{\bowtie} F$ allows to represent a situation in which there is a *cooperation* or *synchronization* between the components E and F. This occurs over an *cooperation set* L that determines how the interaction between the components happens. The set L contains all actions through which the components cooperate. The cooperation set is essential to define the correct behavior of the cooperation: given the cooperation sets H and K with $H \neq K$ then $E \underset{H}{\bowtie} F \not\equiv E \underset{K}{\bowtie} F$.

Unlike the competitive choice, shown above, in the cooperation scope all the components have their own private resources and evolve independently with any activity that not in L. These activities are named *individual*, in contrast with the *shared* activities that are included in L. The cooperation operator behaves as expressed by the following rules:

$$[\text{Coo}_1]\ \frac{E \xrightarrow{(\alpha,r)} E'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,r)} E' \underset{L}{\bowtie} F}\ (\alpha \notin L) \qquad [\text{Coo}_2]\ \frac{F \xrightarrow{(\alpha,r)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,r)} E \underset{L}{\bowtie} F'}\ (\alpha \notin L)$$

$$[\text{Coo}_3]\ \frac{E \xrightarrow{(\alpha,r_1)} E' \qquad F \xrightarrow{(\alpha,r_2)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,R)} E' \underset{L}{\bowtie} F'}\ (\alpha \in L)$$

where the rate R of the action type α in the [Coo$_3$] rule is defined by:

$$R = min(r_\alpha(E), r_\alpha(F)) \cdot \frac{r_1}{r_\alpha(E)} \cdot \frac{r_2}{r_\alpha(F)}$$

and $r_\alpha(E)$ and $r_\alpha(F)$ are the *apparent rates* of the components E and F respectively. The apparent rate will be explained more in detail in section 2.2.2. When the cooperation set $L = \emptyset$, the synchronization $E \underset{L}{\bowtie} F \equiv E \underset{\emptyset}{\bowtie} F$ behaves exactly as the Ccs parallel compsition [12, 19]. The components does not interact each other but proceed independently. In this situation is possible to simplify the notation by substituting the *parallel composition* operator $\|$.

Hiding

The *hiding* or *restriction* operator offer the possibility to hide some aspects of the behavior of a components to external observers. Typically, when an activity finish, an external observer is able to see its action type and duration. The duration is simply the length of time from the completion of the previous activity. In the case that the activity is hidden (the action type is in the set L), the observer can still see the activity duration but is not able to understand the action type. In this situation the action type is said *unknown* and is indicated with τ. Thus the behavior of the hiding operator is the following: the component $E \setminus L$ behaves exactly as E but all the activities within the set L appears as τ to an external observer (or component). This situation is formalized by the two following Sos rule:

$$[\text{Res}_1]\ \frac{E \xrightarrow{(\alpha,r)} E'}{E \setminus L \xrightarrow{(\alpha,r)} E' \setminus L}\ (\alpha \notin L) \qquad [\text{Res}_2]\ \frac{E \xrightarrow{(\alpha,r)} E'}{E \setminus L \xrightarrow{(\tau,r)} E' \setminus L}\ (\alpha \in L)$$

In general, from the other components, an unknown activity τ is treated as an internal delay of a component.

Constant definition

The behavior of the *constant definition operator* is quite simple. Its definition is: given the components A and E and defined the equation $A \stackrel{def}{=} E$, the component A behaves exactly as E. This construct is typically used to rename a component and to construct

recursive definitions. The only assumption is that the set of constants is countable. The structural operational semantic rule is:

$$[\text{Con}]\ \frac{E \xrightarrow{(\alpha,r)} E'}{A \xrightarrow{(\alpha,r)} E'}\ \left(A \stackrel{def}{=} E\right)$$

2.2.2 Passive activities and apparent rate

In this section we give a brief introduction of two important concept of the Pepa language: the *passive activities* and the *apparent rate*. Talking about the hiding operator in the previous section, we have seen that may exist some activities with an unspecified action type τ. A similar thing is possible also with the action rate [12].

Sometime may happen that a component is forced to cooperate with another component. In this case, it is said *passive* with respect to an action type. This means that the activity must be shared with another component that determine the rate of the activity. If a component simultaneously enables more than one activity with a specific action type, is necessary to assign a (probabilistic) weight to each unspecified activity rate. This, with the aim to determine the probabilities of the possible result of the activities.

$$P \stackrel{def}{=} (\alpha, w_1\top).P + (\alpha, w_2\top).Q$$

With regard to the apparent rate, it shall be discussed always with reference to a specific action. In fact, $r_\alpha(P)$ define the total capacity of a component P to carry out activities of type α. The following rules define the apparent rate of each operator of the Pepa language:

$$\begin{aligned}
r_\alpha((\beta,r).P) &= \begin{cases} r & \text{if } \beta = \alpha \\ 0 & \text{otherwise} \end{cases} \\
r_\alpha(P+Q) &= r_\alpha(P) + r_\alpha(Q) \\
r_\alpha\Big(P \underset{L}{\bowtie} Q\Big) &= \begin{cases} min(r_\alpha(P), r_\alpha(Q)) & \text{if } \alpha \in L \\ r_\alpha(P) + r_\alpha(Q) & \text{otherwise} \end{cases} \\
r_\alpha(P \setminus L) &= \begin{cases} r_\alpha(P) & \text{if } \alpha \notin L \\ 0 & \text{otherwise} \end{cases} \\
r_\alpha(A) &= r_\alpha(A) \qquad \text{if } A \stackrel{def}{=} P
\end{aligned}$$

2.3 Underlying Stochastic Model

A PEPA model consists of a set of components that interact according to the rules of the language. The SOS of the language, defined in section 2.2.1, can be used to build a *derivation graph* of a given PEPA model. The derivation graph describes all the possible evolutions of each component of the model. In this multigraph, the component that provide the definition of the model is placed as initial node. Every other component is a node itself. The edges represent the possible transitions between them (activities they can enable) and they are labelled with all the information about the enabled activity (action type and action rate). Given the derivation graph, it may be used to represent the model as a stochastic process.

2.3.1 Generation of the Markov process

The approach followed to generate underlying stochastic process starting from the derivation graph, is quite simple. Let C a finite PEPA model and assume finite, the number of nodes in the derivation graph, then the underlying stochastic process is derived [12] by:

1. Associating a state s_i with $i \in \mathbb{N}^+$, to each node of the derivation graph.
2. Considering each edge as a transition $s_i \longrightarrow s_j$ between the states.

Every activity durations is exponentially distributed, thus the total transition rate between s_i and s_j will be the sum [25] of the rates found on the corresponding edge in the derivation graph. A CTMC is usually expressed as an infinitesimal generator, $\mathbf{Q}$ as seen in section 1.3.1. This elegant representation allows us to express the chain in terms of the rates at which transitions occur. Let C_i and C_j two components of a PEPA model, the transition rate between them can be seen in two different ways: from PEPA point of view, it correspond to the rate at which the model changes its behavior. It stops to behave as C_i and starts behave as C_j. From Markov chain point of view, it is the rate at which a transition between C_i and C_j occurs. In both the cases, the transition rate $q(C_i, C_j)$ is equal to the sum of the activity rates:

$$q_{ij} = q(C_i, C_j) = \sum_{\alpha \in \mathcal{A}ct(C_i|C_j)} r_\alpha$$

where $\mathcal{A}ct(C_i|C_j) = \{\alpha \in \mathcal{A}ct\,(C_i) \mid C_i \xrightarrow{\alpha} C_j\}$ and $i \neq j$. The q_{ij} are the off-diagonal elements of the infinitesimal generator matrix. The last step to complete the construction of the matrix, is to define the elements q_{ii}. They are defined [12] as:

$$q_{ii} = -q(C_i) = \sum_{\alpha \in \mathcal{A}ct(C_i)} r_\alpha$$

where $q(C_i)$ is the rate at which the component C_i stops to behave as the component C_i. Thus the generator is complete. It is easy to see how a PEPA model can be mapped into a CTMC. The conditions for the existence of an equilibrium distribution for a Markov process, are well-known: a stationary probability distribution exists for every time-homogeneous irreducible Markov process [12]. However, all PEPA models are time-homogeneous since the rate of activities are independent of time. Once the CTMC is defined, it is possible to derive its steady-state solution by exact or approximate techniques.

2.3.2 Kronecker representation

A PEPA model is described as an interaction of components that perform actions and that cooperate with each other. The cooperation of the components and the size of the models give rise to an exponential growth in the number of states. A well-known solution to represent the states of underlying Markov process is to express a PEPA model in terms of a Kronecker product of terms. The Stochastic Automata Networks (SAN), proposed by Plateau in [22], consists of a number of individual stochastic automata that can operate independently of each other. Plateau was able to prove that the global generator matrix of the Markov process underlying a SAN, can be analytically represented using the Kronecker algebra. In [13], the authors shown that it is also possible analytically represent a PEPA model using the Kronecker algebra by building an automatic translation system, analogous to the SAN representation. The infinitesimal generator matrix of a complete model may be represented as a sum of tensor products:

$$Q = \bigoplus_{i=1}^{k} R_i + \sum_{\alpha \in \mathcal{S}} r_\alpha \left(\bigotimes_{i=1}^{k} P_{i,\alpha} - \bigotimes_{i=1}^{k} \bar{P}_{i,\alpha} \right) \tag{2.1}$$

where $\bigotimes$ and $\bigoplus$ are the tensorial sum and product operators respectively, k is the number of components of PEPA system and $\mathcal{S}$ is the set of cooperating action. The matrix R_i is the transition probability matrix of a component C_i relating to its individual activity, the rate r_α is the minimum of the functional rates of an action α over all components C_i with $i = 1, \ldots, k$ of the system and, in the end, we also have the probability transition matrix $P_{i,\alpha}$ of a component C_i related to activity of type α. The matrixes $P_{i,\alpha}$ are not generators, so we need to introduce a normalization matrix $\bar{P}_{i,\alpha}$, to ensure that row sums are zero.

The idea behind the equation (2.1) is that a model is composed by interacting and non-interacting components. In a non-interacting component C_i, at least one activity is non-shared. To C_i, we associate a generator matrix R_i. The resulting matrix represents the local transitions of C_i. In an interacting component, we associate a transition probability matrix $P_{i,\alpha}$ with each action type α in the set of cooperating actions. The matrix detects if a component is able to participate in the shared activity. In this way, each element of the matrix, represents the transition probability of a component C_i with activity α and rate $r_\alpha(C_i)$.

2.3.3 Performance measures derivation

The most direct way to compute the quantitative behavior of PEPA models, is to apply a reward structure [15] to the states of the underlying Markov chain. A reward structure consist in a set of reward functions $\rho\colon \mathcal{S} \to \mathbb{R}^+$, where ρ is a function on the states of the Markov process associated with the model. A generic reward R is computed as follows:

$$R = \sum_k \pi_k \cdot \rho(C_k) \tag{2.2}$$

Basically, what the formula does, is to accumulate bonus as long as a process remain in a state and then collect the reward when the process change states. Using the reward R the performance measures such as utilization, throughput, population level etc. can be directly derived starting from the steady state distribution $\boldsymbol{\pi}(\cdot)$.

2.4 Conclusion

In this chapter, we gave a quick introduction of the Performance Evaluation Process Algebra PEPA, an extension of classical process algebras that introduces a rate for the transitions and the concept of synchronization and cooperation between processes. The main strength of PEPA lies in the exponential distribution of the rates at which transitions between states occur such that, it can easily be considered as a continuous-time Markov chains. We also shown, how it is possible to build the underlying Markov chain of a generic PEPA by the tensorial representation. Finally, we highlighted the simplicity of the performance indexes derivation process through the analysis of the steady-state probability distributions of the chain.

Part II

Contributions

Chapter 3

Model definition

3.1 Introduction

In the previous chapters we described all the tools that help us in the definition of the model. We move now into the heart of this thesis. The main goal is to present the formal model to compute the energy performance of frequency regulatory policies.

What we essentially do, is to formalize in PEPA language two interacting models and to compute the performance indexes through the structures provided by the process algebra. The first model describes, in terms of operational frequency, the different levels of workload of a generic CPU. The second, describes its allowable working levels, always in terms of operational frequency. The interaction between models is obtained through the cooperation operator provided by PEPA language. The analysis of the obtained data will allow us to determine whether the proposed model represents a valid policy to optimize the power consumption of CPU.

The chapter is structured as follows: in section 3.2 we will give the definition of the two models, explaining in detail the choices in the development of the *Workload model* (3.2.1) and of the *Frequency model* (3.2.2). In section 3.3 we will define the relevant performance indexes, focusing in particular on the throughput (3.3.1) and on the power consumption and loss (3.3.2).

3.2 Power management model

Dynamic voltage and frequency scaling (DVFS) is a famous technique used to automatically reduce the power consumption in a very large set of heterogeneous devices. It is able to adjust the power consumption of a modern computer microprocessor by decreasing its operative frequency. Significative applications of the DVFS technique can be found in the reduction of power consumption in large data centers but also in mobile devices such as tablets and smartphones [9, 6], where energy is provided by a battery and thus is limited. Intuitively, this technique reduces the number of instructions a processor can execute in a given amount of time, thus reducing performance in terms of throughput. For this reason,

the central question is on how to build a model to manages the power consumption by optimizing the tradeoff between workload and frequency of the CPU.

The process algebra PEPA helps us in the representation of this situation. We will formalize a 4-states model to express the workload of the CPU and a k-state model, with $k \in \{2, 3\}$, to represent the processor operative frequency levels (or *p-states* according with the ACPI specification [1]). The two models can communicate via cooperation operator (2.2.1). A check is performed on the workload level with rate r and the CPU state is updated.

3.2.1 Workload processor model

The workload model WL consists of four different states. Each state is labelled WL_i where $i = 1, \ldots, 4$ and represents a different level of workload of the CPU in terms of operational frequency. From one of these states, the model has two different possibilities: it may loop to the same state with an action rate r, enabling a shared activity of type w_i or it can enable an individual activity and jump to any other state. This last activity has an action type x and an action rate which is determined by the infinitesimal generator $\mathbf{Q}$ of the Markov chain that describe the behavior of the workload. The model is easily described from the following figure:

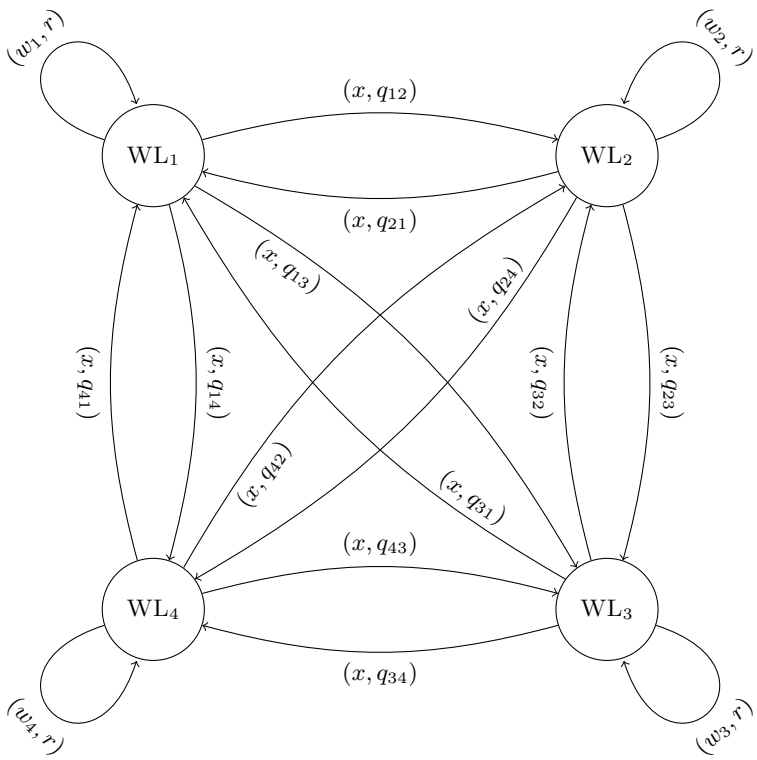

Figure 3.1: The four states model to represent the workload of a CPU.

At each time instant, the model can be in one and only one state. The graphical rep-

resentation given by figure (3.1) can be directly translated in PEPA language. The model WL is defined as the competitive choice of the four components WL_i. Thus $WL \stackrel{def}{=} WL_1 + WL_2 + WL_3 + WL_4$ and each component defined as:

$$\begin{array}{lcccccccc}
WL_1 & \stackrel{def}{=} & (w_1, r).WL_1 & + & (x, q_{12}).WL_2 & + & (x, q_{13}).WL_3 & + & (x, q_{14}).WL_4 \\
WL_2 & \stackrel{def}{=} & (x, q_{21}).WL_1 & + & (w_2, r).WL_2 & + & (x, q_{23}).WL_3 & + & (x, q_{24}).WL_4 \\
WL_3 & \stackrel{def}{=} & (x, q_{31}).WL_1 & + & (x, q_{32}).WL_2 & + & (w_3, r).WL_3 & + & (x, q_{34}).WL_4 \\
WL_4 & \stackrel{def}{=} & (x, q_{41}).WL_1 & + & (x, q_{42}).WL_2 & + & (x, q_{43}).WL_3 & + & (w_4, r).WL_4
\end{array}$$

As already discussed, the four components are implicitly competing for the same resource. The first of them that complete the activity is allowed to use the resource, the other are discarded. The activities which action type is x, are the individual activities. Its rates are determined by the elements q_{ij} of the infinitesimal generator. The other activities have an action type w_i and allow the cooperation between the two models.

3.2.2 Frequency processor model

The frequency model FR, has been formalized in two different ways. In order to evaluate how the behavior of the model changes when the number of frequency levels increases, we developed a 2-states and a 3-states CPU. The first one consists of a simple processor with an *high speed* level and a *low speed* level. Each state is labelled FR_i with $i \in \{1, 2\}$ and represent one of this two situations. A low frequency level is assigned to the first state and an high level to the second. This behavior can be seen in the following figure:

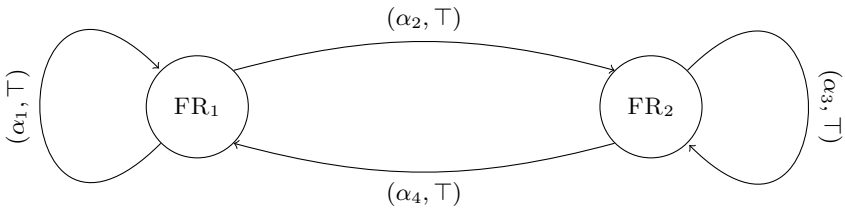

Figure 3.2: The two states model to represent the speed of a CPU.

There are some important aspects to highlight. The first is that the activity strictly depends on the the behavior of the workload and on the number of the states of the WL model. The previously defined workload model has four states and four shared activities. Those same activities must be found in the CPU model. The second is that the given formalization of the model FR, it's just a generic formalization. All the action types are

labelled with a general α_i, this leaves the possibility to define different behaviors of the model.

Consider, for example, a CPU in which the state FR_1 responds to shared actions w_1, w_2 and w_3 and the state FR_2 only to the action w_4. Translating the foregoing in simple words, we could say that when an action type w_1, w_2 or w_3 occurs in the workload model, the frequency model remains in the state FR_1. When an action type w_4 occurs, it jumps to the state FR_2. In this case, α_1 is the set of activities to which the state FR_1 responds and $\{w_1, w_2, w_3\} \in \alpha_1$. Likewise, α_2 is the set of activities to which the state FR_2 responds and $\{w_4\} \in \alpha_2$. The translation of the model in the process algebra becomes $FR \stackrel{def}{=} FR_1 + FR_2$ where:

$$\begin{array}{lcllllll} FR_1 & \stackrel{def}{=} & (w_1, \top).FR_1 & + & (w_2, \top).FR_1 & + & (w_3, \top).FR_1 & + & (w_4, \top).FR_2 \\ FR_2 & \stackrel{def}{=} & (w_1, \top).FR_1 & + & (w_2, \top).FR_1 & + & (w_3, \top).FR_1 & + & (w_4, \top).FR_2 \end{array}$$

Therefore, is easy to see how it is possible to aggregate in different ways the activities in order to give a different behavior of the model. The third important aspect is that the activities are passive and as seen in section 2.2.2, their action rate is set to $\top$. According to this, the transitions in the model FR, depend on what happens in the model WL. The rate at which an action type w_1 occurs in the model FR depends on the rate of action type w_1 in the model WL. So, all the activities are passive and the weights w_i are omitted because every activity has the same probability to be enabled.

At the end of all these considerations, the figure 3.2 can be finally translated into PEPA language. The frequency model is defined by the competitive choice $FR \stackrel{def}{=} FR_1 + FR_2$ where:

$$\begin{array}{lcllll} FR_1 & \stackrel{def}{=} & (\alpha_1, \top).FR_1 & + & (\alpha_2, \top).FR_2 \\ FR_2 & \stackrel{def}{=} & (\alpha_3, \top).FR_2 & + & (\alpha_4, \top).FR_1 \end{array}$$

By following the same approach, the second model has been formalized. The 3-states frequency model is constituted by a *low speed* level (state FR_1), a *medium speed* level (state FR_2) and a *high speed* level (state FR_3). The states are still labelled FR_i with $i \in \{1, 2, 3\}$ and represent one of this three different frequency levels of the CPU.

The increase in the number of states, often means giving to the model a better chance to find an optimal state. A state in which the tradeoff between the power consumption and throughput of the system is excellent. However, the increase in the number of the states is not the only possible approach. A better policy in the transitions between the states can improve the power consumption with minimal impact on the system performances. For all these reasons, we choose to build a frequency model that attempts to maintain an acceptable level of throughput. This is obtained by developing a policy that forces the model to reach immediately the *high speed* level and then gradually move down to the

slow speed level by passing for the intermediate state. In the following figure this policy is clear:

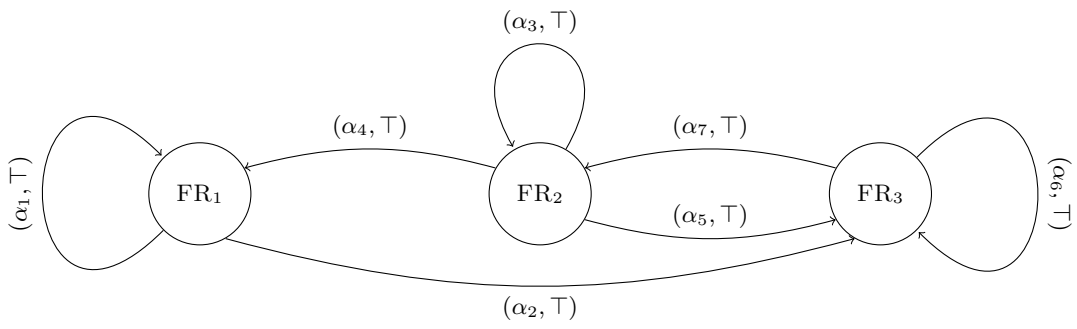

Figure 3.3: The three states model to represent the speed of a CPU.

We see indeed, as the state FR_1 can not directly pass to the state FR_2 but is forced to reach the state FR_3 where the operating frequency is maximum. Similarly, the state FR_3 can not jump directly to state FR_1 but is forced to pass through FR_2 by performing a one-step jump. The frequency model is now translated in PEPA language. It corresponds to the competitive choice $FR \stackrel{def}{=} FR_1 + FR_2 + FR_3$, with:

$$\begin{array}{lcllll}
FR_1 & \stackrel{def}{=} & (\alpha_1, \top).FR_1 & + & (\alpha_2, \top).FR_3 & \\
FR_2 & \stackrel{def}{=} & (\alpha_3, \top).FR_2 & + & (\alpha_4, \top).FR_1 & + \quad (\alpha_5, \top).FR_3 \\
FR_3 & \stackrel{def}{=} & (\alpha_6, \top).FR_3 & + & (\alpha_7, \top).FR_2 &
\end{array}$$

As in the 2-states model, also in this, the activities are passive. Thus, the transitions, do not occur at a fixed action rate but are linked to the behavior of the workload model. Furthermore, many aggregations of the states can be chosen according to the workload input. The activities can be aggregated in different combinations to balance the model. If, in fact, we are in presence of a medium-high workload, will be more appropriate to aggregate the actions towards the *high speed* state. In this way, α_1 is the set of activities to which the state FR_1 responds and $\{w_1\} \in \alpha_1$, α_3 is the set of activities to which the state FR_2 responds and $\{w_2\} \in \alpha_2$ and finally α_6 is the set of activities to which the state FR_3 and $\{w_3, w_4\} \in \alpha_3$. All other transition behave then accordingly.

3.2.3 The rate question

Before moving our attention to the definition of relevant performance indexes, we focus on the parametrization of the model. We said that the four shared activities w_i with $i = 1, \ldots, 4$, of the model WL, occur with a rate equal to r. Ideally, this parameter, can be thought as a observation window, a period in which the workload is analyzed. Upon close of the window, the model enables an activity. The main problem, is that within

this period, the behavior of the workload may change dramatically and very quickly. Therefore, the size of the window must be chosen carefully. A window too large (just a few observations per unit of time) does not allow to capture accurate information on the behavior of workload. For example, special situations such as sudden changes in the workload, would be completely ignored. On the other side, a smaller observation window (many observations per unit of time), would allow to collect much more detailed information on the workload, but the payload, generated by the frequent change of level, would degrade significantly the performance. In the following figure we show an example of this problem:

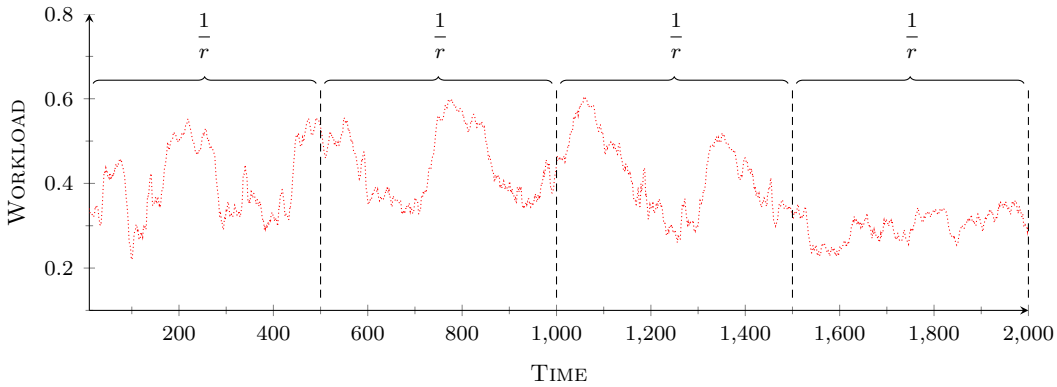

Figure 3.4: Example of workload trace (dotted line) and a too large observation window (separated by dashed lines).

Considering, for example, the third period, we can clearly see that two peaks in the workload level occur within it. However, when the window closes, the detected level is very low. The opposite situation occurs when the first window closes. The detected level is high, but it is evident that within the observation window there are some moments in which would possible to reduce the frequency of the processor. A different situation there is in the fourth period in which the load level remains constant and thus the detection is correct.

So, the inappropriate choice of the window size (and therefore a rate r too small or too large) affects the correctness of the detection of the workload level. For these reasons, we chose to test the model with different settings of parameter r and compared the results with an ideal analytical model in which $r \to \infty$.

3.3 Performance indexes computation

In section 2.3.3, we explained how it is possible to directly compute the performance indices of a Pepa model from the steady-state probability distributions. We also defined the concept of reward function to define these measures. In this way, the required performance indexes may be expressed in terms of some identifiable aspect of system behavior. This behavior is strongly connected with the activities which the components can enable. Thus, the performance indexes can be defined by associating a reward function with an activity

or set of activities. Let's see now how to define the performance indexes of throughput and power consumption of the developed model.

3.3.1 Throughput

Taking the original definition of throughput [12] of a PEPA model, it is associated with a specific action type. In general, it is defined as the average number of activities of a particular type completed per unit of time. From this definition, the reward function of the throughput is directly derived from the population level of the state that can perform a specific action type α and from the rate of the action itself.

We followed a slightly different way to compute the throughput. Let $T_{WL}(t)$ and $T_{FR}(t)$ be the evolution of the level of throughput over the time of the models WL and FR respectively. We consider the throughput of the cooperation $WL \underset{w_1,w_2,w_3,w_4}{\bowtie} FR$ as:

$$T = \frac{1}{t} \cdot \int_0^t min\{T_{WL}(t)\,, T_{FR}(t)\}\ dt$$

As we will explain better in the next chapter, a frequency level is assigned to each state of the two models. Let $fq\colon \mathcal{S} \to \mathbb{R}^+$ a function that assign a frequency value to each state of the models (for example $fq(WL_1) = 25$ is the frequency level of the state WL_1) and let $\pi_{(w,f)}$ the steady-state probability distributions of the PEPA models where $\pi_{(w,f)}$ is the probability that WL model be in state w and FR model be in state f. So, the reward function for the throughput is:

$$\rho(w,f) = \begin{cases} fq(w) & \text{if } fq(w) < fq(f) \\ fq(f) & \text{otherwise} \end{cases}$$

and, the reward structure $R^{(Thr)}$ for the throughput, is defined as:

$$R^{(Thr)} = \sum_{w \in \mathcal{W}} \sum_{f \in \mathcal{F}} \pi_{(w,f)} \cdot \rho(w,f) \tag{3.1}$$

where $\mathcal{W}$ and $\mathcal{F}$ are the sets of the states of the models WL and FR respectively. By using the reward structure $R^{(Thr)}$, the throughput is not not related to a specific activity but depends only by the current state of the two models and, thus, exclusively by the effective frequency level.

3.3.2 Power consumption

The power consumption can be derived, under appropriate assumptions, directly from the frequency. We use a simple model [29, 23] in which the power consumption of a

CMOS corresponds to the sum of the static power consumption P_s and dynamic power consumption P_d. The first one is not related to the switching activity and captures the power consumption of peripheral devices so, we assume it is a constant and do not consider. The dynamic consumption, instead, depends both on the switching activity both on the supply voltage. It can be expressed as:

$$P_d = \alpha \cdot C_L \cdot V^2 \cdot f \tag{3.2}$$

where α is the switching probability, C_L is the load capacitance, V is the supply voltage and f is the operational frequency of the circuit. The supply voltage linearly depends on the operational frequency f and can be rewritten as $V = \beta \cdot f$, resulting in:

$$P_d = \alpha \cdot C_L \cdot (\beta \cdot f)^2 \cdot f = \alpha \cdot C_L \cdot \beta^2 \cdot f^3 \tag{3.3}$$

When the frequency decreases by a factor of $s \geq 1$, we obtain a corresponding decrease in power consumption. The target frequency $f_s = s^{-1} \cdot f$ can be replaced in equation (3.3) and the new power consumption can be easily computed as:

$$\begin{aligned} \bar{P}_d &= \alpha \cdot C_L \cdot \beta^2 \cdot f_s^3 \\ &= \alpha \cdot C_L \cdot \beta^2 \cdot \left(s^{-1} \cdot f\right)^3 \\ &= \alpha \cdot C_L \cdot V^2 \cdot s^{-3} \cdot f \\ &= s^{-3} \cdot P_d \end{aligned} \tag{3.4}$$

Equation (3.4) allows us to define the reward structure. We must, first of all, determine the scale factor. Remembering the equation $f_s = s^{-1} \cdot f$, we need a fixed reference frequency. We might choose to set $f_s = 2000Hz$ and $P_d = 100W$. This is due to we assume the power consumption of a $2.0Ghz$ CPU is equal to $100Watt$ when it works is at its maximum speed. The reward function $\rho(u)$ is then computed as:

$$\rho(u) = s^{-3} \cdot P_d = \left(\frac{fq(u)}{f_r}\right)^3 \cdot P_d = fq(u)^3 \cdot 1.25 \cdot 10^{-4}$$

where u is a generic state of the model FR and $fq(u)$ is the frequency level of the state u. The reward structure is thus:

$$R^{(Pwc)} = \sum_{w \in \mathcal{W}} \sum_{f \in \mathcal{F}} \pi_{(w,f)} \cdot \rho(f) \tag{3.5}$$

A more interesting measure is the power loss of the model. This can be easily derived from the formulas above. The reward structure becomes:

$$R^{(Pww)} = \sum_{w \in \mathcal{W}} \sum_{f \in \mathcal{F}} \pi_{(w,f)} \cdot [\rho(f) - \rho(w)] \cdot \mathbb{1}_{fq(w) < fq(f)} \tag{3.6}$$

where $\mathbb{1}_{fq(w)<fq(f)}$ is the *indicator functions* [16] which takes the value 1 if the condition is true.

3.3.3 Cost function

Let T_r be the throughput, and let P_l be the power waste. In order to evaluate the global performances of the model we define the cost function $C(\alpha, \beta)$ as:

$$C(\alpha, \beta) = \alpha \cdot \frac{P_l}{\max(P_l)} - \beta \cdot \frac{T_r}{\max(T_r)} \tag{3.7}$$

where the coefficients α and β reflect the relative costs of the power loss and throughput, respectively. No assumption concerning the relative magnitude of those costs is made. However, in cases of practical interest, one is likely to have $\alpha < \beta$. The objective of the analysis is to provide expressions for computing T_r and P_w, so that the cost function can be minimized. This will allow us to quickly detect the best energetic policy for any input chain.

3.4 Conclusion

In this chapter, the foundations for all the thesis has been laid. Firstly, we given the definition of models, by describing the workload of a CPU and the frequency levels at which it can operate. Both models have been shown both from a graphical point of view and from the PEPA point of view. We also defined how these models work together through the cooperation operator, made available from the process algebra. Secondly, we formally defined all the performance measure that must be computed in order to evaluate the behavior of the model. In the next chapters we will test the proposed framework by performing a performance analysis of the model, parametrized starting from a set of both synthetic and real data.

Chapter 4

Model validation on a synthetic dataset

4.1 Introduction

Now that the formal model and the relative performance indexes have been defined, the next step is to evaluate their behavior. To do this, we use an algorithm for the automatic formalization in process algebra PEPA and for the subsequent collection of measures.

In this chapter, we will present this algorithm. The inputs are constituted by two different continuous-time Markov chains and by a set of frequency regulatory policies. The chains are specifically made for this stage and represent the CPU workload behavior. In particular, what we want to verify, is the behavior of the model subjected to a workload, both moderate and intense. The policies consist of two and three states models with different states aggregations. Finally, extensive tests on the model will be performed and the results will be presented.

The chapter is structured as follows. In section 4.2 we will present the algorithm, describing all its components and how they work. In particular, we will explain in detail the procedure to automatically generate the PEPA files (4.2.1), the procedure to clean the steady-state probability files (4.2.2) and the procedure to compute the performance indexes (4.2.3). In section 4.3 we will talk about the model parametrization, discussing the CTMCs which characterize the workload of the CPU (4.3.1) and the construction of regulatory policies (4.3.2). Finally, in section 4.4 we will examine the gathered results.

4.2 The algorithm

The main goal of algorithm is to automate many of the procedures required to compute the energy performance of each policy. Unfortunately, not all the procedures can be automated. For example, the computation of the steady-state probability distributions must be manually done. Development, takes as its starting point the numerical computing

environment MATLAB in version 2012A. The strength of MATLAB is in its ability to handle a large amount of data in a simple and efficient way. The algorithm consists of a MATLAB script (A.1) and of a set of suitably developed functions. The computation of probability distribution is performed outside of MATLAB environment and is delegated to the PEPA Eclipse plugin [26]. We can ideally split the script into 5 parts, each with a specific mission: initialization of the data structures, generation of the PEPA models, computation of steady-state probability, files cleanup and, finally, derivation of performance measures.

Initialization

The first part of the script involves the initialization of all the data structures required to the computation. In particular, the initialization procedure includes variables that keep track of the folder (A.1 line 8) which contain the input and output data, of the names of the experiments (A.1 line 10) and of the matrixes representing the different policies.

The initialization of these matrixes is evident in (A.1 lines 11 $\rightarrow$ 16). Their aim is to represent the behavior of FR model. Consider, for example, the matrixes:

$$\alpha = \begin{pmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \end{pmatrix} \qquad \delta = \begin{pmatrix} \delta_{11} & \delta_{12} & \delta_{13} & \delta_{14} \\ \delta_{21} & \delta_{22} & \delta_{23} & \delta_{24} \\ \delta_{31} & \delta_{32} & \delta_{33} & \delta_{34} \end{pmatrix}$$

They provide the representation of possible transitions for the two and three states model. Each policy is labelled with a letter from the greek alphabet. The rows are the current state of the model and the columns are the received action type. Hence, the element α_{13} means that the frequency model, jumps from the state FR_1 to the state $FR_{\alpha_{13}}$ when the workload model enables an action type w_3.

The script also initializes the variables `r` and `map`. First is used to build the infinitesimal generator of the CTMC that characterizes the workload. The meaning of the `map` variable is a little bit more complicated. It will be mainly used in the next chapter, when we will perform the fitting of a real workload trace in a Markov process. In the construction of the infinitesimal generator might happen that some states are never reached. This situation can not be ignored because the Markov chain underlying the PEPA model would no longer be ergodic. For this reason, we exclude them from generator. The `map` reminds of the original position of these states, since, maintain the order of states is essential for the correctness of whole process.

Models generation

In the second part of the script, files containing the formalization of the PEPA models are generated. These, are directly interpretable by the PEPA Eclipse plugin. As mentioned, the formalization process is executed by the function `pepa_generator` whose code can be seen in (A.4). This procedure, takes as inputs the infinitesimal generator which characterizes the workload of CPU and an above described matrix. The generation process and files anatomy will be analyzed more in detail in following sections.

Steady-state probability computation

The third part concerns the steady-state probability distribution computation of the models subjected to the shared activities. The script waits until all the computed data are made available. As explained, the analysis is performed outside of MATLAB. It is delegated to the PEPA Eclipse plugin and can not be automated. What is done by the plugin is quite simple. We summarize the whole procedure in the following screens:

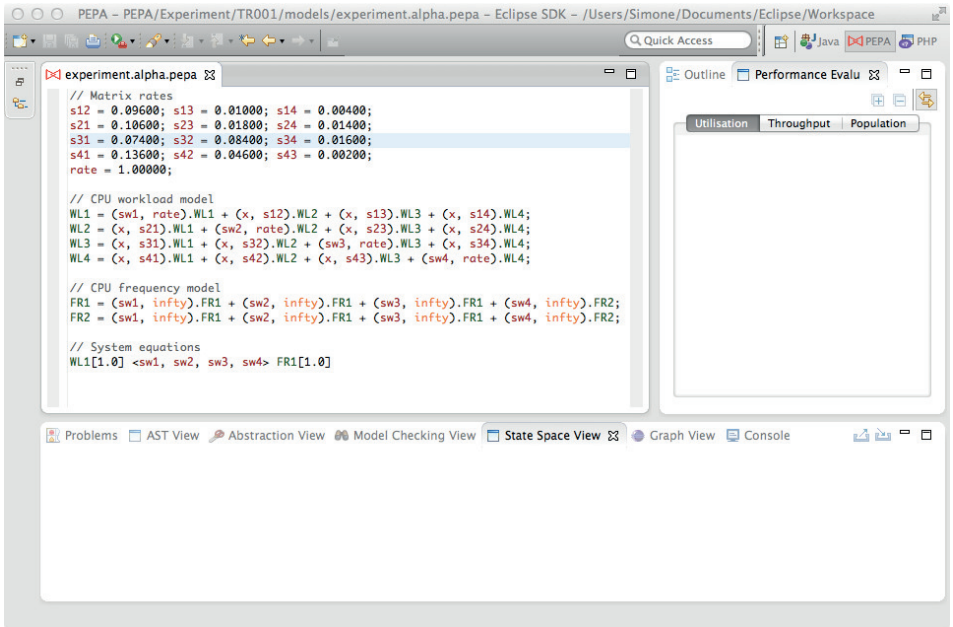

Figure 4.1: Files opening. The `.pepa` *files are directly interpretable by* PEPA *Eclipse plugin.*

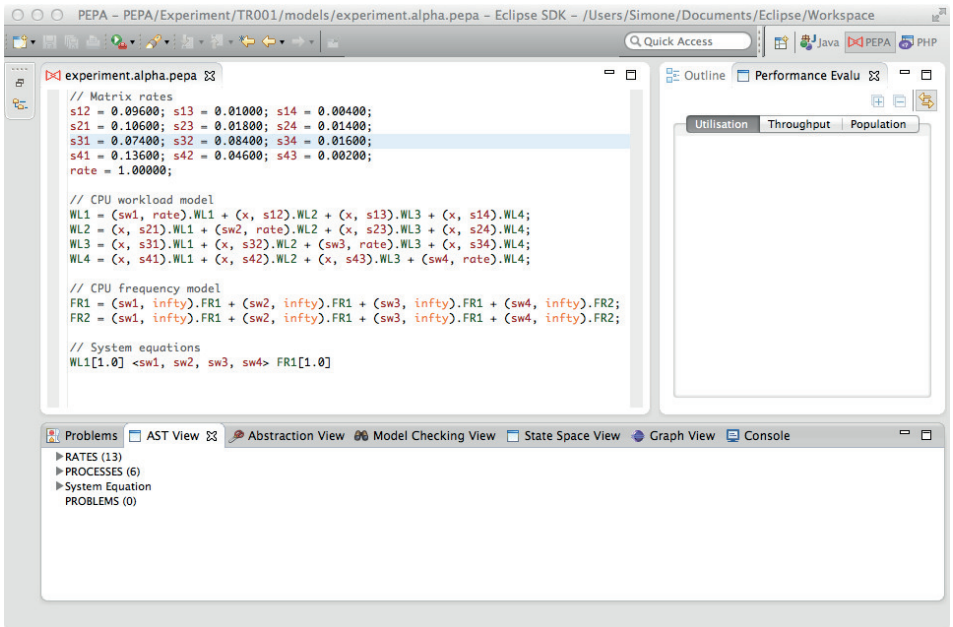

Figure 4.2: Before compute the steady-state probability distribution, the whole model must be derived (under PEPA $\longrightarrow$ CTMC $\longrightarrow$ DERIVE *menu).*

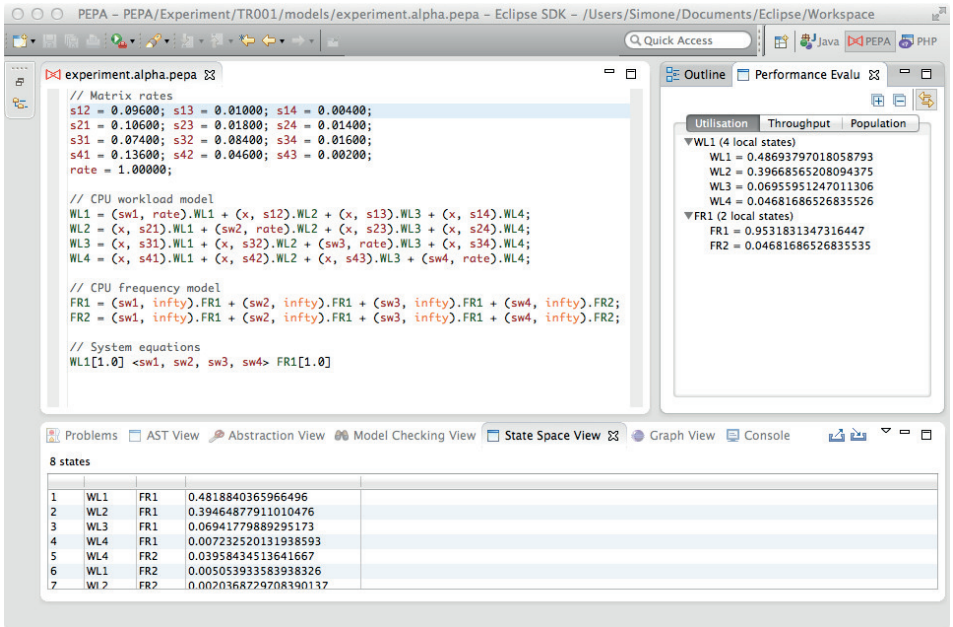

Figure 4.3: Once the model has been derived, it is possible to perform the steady-state analysis (under PEPA ⟶ CTMC ⟶ STEADY STATE ANALISYS *menu).*

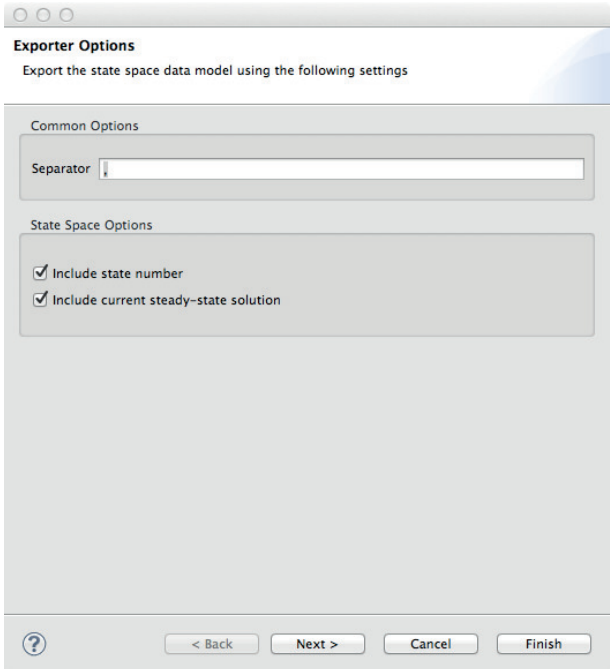

Figure 4.4: Finally, the files can be exported in `.statespace` *format.*

In addition, the PEPA Eclipse plugin offers many other useful features. A text editor is available to write and edit the `.pepa` files. Once saved, the input is parsed and, if the parsing is successful, static analysis is carried out. All the detected warning and errors are shown in the *properties view*. The *performance evaluation view* shows information about throughput, utilization and population level. This view is automatically updated when the model in the editor is solved. Finally, the plugin also supports *time-series analysis* by stochastic simulation or solving ODEs and it has a interface for abstracting and model checking *Continuous Stochastic Logic* (CSL) properties.

Steady-state probability data cleanup

The cleaning of the steady-state files exported from the plugin, is what is responsible the fourth part of the script. The procedure is performed by the `probability_clean` function, developed specifically to carry out this operation. From the PEPA Eclipse plugin is possible to save (figure 4.4) the computed data as `.statespace` file. The following code provides an example of the format of these files:

```
1,{WL1,FR1},0.43291425816149043
2,{WL2,FR1},0.3529987302856876
3,{WL3,FR1},0.0584867795397096
4,{WL4,FR1},0.03922385427464403
5,{WL3,FR2},0.011072732930402944
6,{WL4,FR2},0.007593010993711244
7,{WL1,FR2},0.05402371201909776
8,{WL2,FR2},0.043686921795256455
```

Listing 4.1: Example of `.statespace` file exported from the PEPA Eclipse plugin.

These files are not directly interpretable by MATLAB. For this reason, they are cleaned and converted into a more convenient CSV format. Then they are prepared for subsequent analysis. A CSV file is easily interpretable by MATLAB that natively supports the reading and writing operations. The function also performs the construction of the probability matrix PM. This contains the joint steady-state probability of the WL and FR models. Thus, a generic element PM_{ij}, indicates the probability to be at the same time in state i of the workload model and in state j of the frequency model. The procedure will be described more in detail in section 4.2.2.

Performance computation

The last part of the script concerns the computation of performance measures. We have already discussed in section 2.3.3, the performance measures derivation process through the application of reward structures. In general, these structures are given in the form:

$$R = \sum_k \pi_k \cdot \rho(C_k)$$

By the application of the reward functions $\rho(C_k)$ to the probability matrix PM defined above, is therefore possible to obtain the measures. This procedure is executed by the

`model_function_performance` function (A.6) described below. The output of the function are the matrixes TR, PC and PL that respectively contains the throughput, the power consumption and, last but not least, the power loss.

In the following sections, we will give a more detailed description of the functions used by the MATLAB script just defined.

4.2.1 Generation of the models

Procedure `model_function_pepa_generator` (A.4), is responsible for the production of the files which contain the formalization of the models in process algebra. These files must be syntactically correct in order to be directly interpreted by PEPA Eclipse plugin. The function is parameterized to handle different kinds of situations. It takes as input the infinitesimal generator `Q` that characterizes the behavior of the workload of the CPU, a matrix `L` representing the behavior of the frequency policy, the variable `map` to keep track of the original position of the deleted states of the final infinitesimal generator, the number `k` of the states of WL model and, finally, the variable `filename` which is the name of the output file.

```
// Matrix rates
q12 = 0.09600; q13 = 0.01000; q14 = 0.00400;
q21 = 0.10600; q23 = 0.01800; q24 = 0.01400;
q31 = 0.07400; q32 = 0.08400; q34 = 0.01600;
q41 = 0.13600; q42 = 0.04600; q43 = 0.00200;
rate = 1.00000;

// CPU workload model
WL1 = (w1, rate).WL1 + (x, q12).WL2 + (x, q13).WL3 + (x, q14).WL4;
WL2 = (x, q21).WL1 + (w2, rate).WL2 + (x, q23).WL3 + (x, q24).WL4;
WL3 = (x, q31).WL1 + (x, q32).WL2 + (w3, rate).WL3 + (x, q34).WL4;
WL4 = (x, q41).WL1 + (x, q42).WL2 + (x, q43).WL3 + (w4, rate).WL4;

// CPU frequency model
FR1 = (w1, T).FR1 + (w2, T).FR3 + (w3, T).FR3 + (w4, T).FR3;
FR2 = (w1, T).FR1 + (w2, T).FR2 + (w3, T).FR3 + (w4, T).FR3;
FR3 = (w1, T).FR2 + (w2, T).FR2 + (w3, T).FR3 + (w4, T).FR3;

// System equations
WL1[1.0] <w1, w2, w3, w4> FR1[1.0]
```

Listing 4.2: Example of `.pepa` file generated by the `PEPAmodelGenerator` function. This code is directly interpretable by the PEPA *Eclipse plugin.*

The translation process is divided in four steps, as many as the different parts of which a `.pepa` file is made. First of all, are indicated all action types and action rates of the activities. They correspond to the rate of the infinitesimal generator for all the activities whose action type is x and to the rate r for the activities with action type w_i. Then models

are built. The WL model is generated by a double `for` loop that performs a complete visit of the infinitesimal generator matrix. When the current element is a diagonal element of the generator, the function replaces the action type x with the action type w_i and rate r. The model FR is generated in the same way from the matrix L that already contains all the necessary information about the policy. Finally, the so-called *system equations* are generated. They provide to models all the informations on the cooperation. In particular, they explicitly indicate the set L of shared actions that allow the synchronization.

4.2.2 Cleaning of the steady-state probability files

At this stage, we perform the cleaning of the files resulting by the analysis of the PEPA Eclipse plugin. The function `model_function_probability_clean` takes as input the name of the file to be cleaned and the map of the states deleted from the final infinitesimal generator. The process is quite simply. A `.statespace` file consists of a CSV file, but its structure does not allow to use it directly by MATLAB. In particular, looking the listing 4.1, each row provides the informations about the state of the WL and FR models as well as their probability. The problem is on the format of the second column. The strings `{WL`, `FR` and `}` must be removed in order to use the numerical values as the indexes of a matrix. The algorithm takes care to remove this information and then builds the matrix PM which contains all the probabilities.

4.2.3 Computation of the performance indexes

The `model_function_performace` function surely performs the most delicate task. It is, in fact, responsible of the computation of the reward structures. The inputs of the function are only three: the previously defined `PM` matrix, the variable `e` which indicates the name of the experiment, and the number `s` of the states of the WL model. The key point of the function is to assign the frequency levels to each state of the model. As well as the four states of the workload model take respectively the values $[500, 1000, 1500, 2000]$ which indicate the frequency level, even to the states of FR is assigned a frequency level, specific for each policy. Once the parametrization is complete we can apply the chosen reward functions. The output is constituted by the matrixes TR, PC and PL that contain the energy performance measures of each policy.

4.3 Model parametrization

The parameterization of the model is performed under different aspects, distinct but important in equal measure. The first aspect concerns the rate at which a shared activity in the WL model is enabled. In section 3.2.3, we talked about how much important it is to carefully choose this rate. The experiments will be performed with six different values of the parameter r. After a few trials we discovered that the behavior of the model tends to stabilize for values greater than 10^3. According to this result, we evaluate the model on the

set of values $r \in [10^{-2}, 10^{-1}, 10^{0}.10^{1}, 10^{2}, 10^{3}]$, which offer a wide margin of analysis. The second aspect regards the characterization of the behavior of the workload. As explained, it entirely depends on the infinitesimal generator of a CTMC. The parameterization is obtained, in this case, through the construction of two empirical chains that represent different workload situations. Finally, the third aspect involves the definition of a set of frequency regulatory policies. As we know the behavior of the model is determined not only by the number of frequency levels but also on how transitions occur between states. Developed policies include both aspects, by defining regulatory policies with two and three frequency levels and by aggregating the activities in different ways. The following map gives an idea of the whole parametrization process:

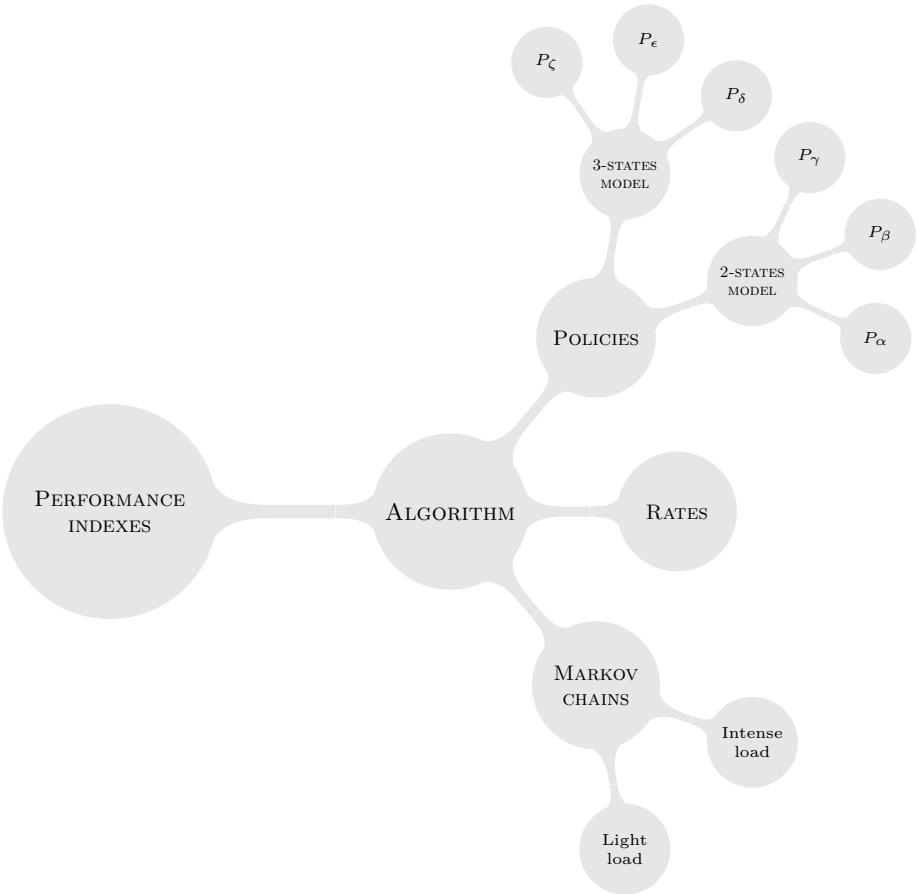

Figure 4.5: The map of the parametrization process shows how it involves many different aspects.

4.3.1 Syntethic traces

In order to express the characterization of the workload, the easiest way is probably develop a CTMC by the transition probability matrix and then derive the infinitesimal generator. What we want to do, is to ignore the time spent in any state of the chain and to consider only the sequence of transitions. Therefore, we define a discrete-time Markov chain (DTMC), $\{E_k : k \in \mathbb{N}\}$ on a state space $\mathcal{S}$ with transition probability matrix $K = [k_{ij}]$ and $i, j \in \mathcal{S}$. Let $\{T_k : k \in \mathbb{N}\}$ be some homogeneous Poisson process [5, 17, 25]

with intensity $\lambda > 0$ and counting process $\{N_t : t \geq 0\}$. Then, the process $\{X_t : t \geq 0\}$ is called *uniform Markov jump process* with $X_t = E_{N_t}$ under the assumption of independence of N_t and E_k. The transition probability matrix of this process is defined by the equation:

$$\mathbf{P}(u) = \sum_{n=0}^{\infty} e^{-\lambda u} \cdot \frac{(\lambda \cdot u)^n}{n!} \cdot \mathbf{K}^n = e^{u\lambda(\mathbf{K}-\mathbf{I})} \tag{4.1}$$

Now, recalling the *Kolmogorov's forward equation* (1.13) and putting together with the equation (4.1) we obtain the infinitesimal generator, given by:

$$\mathbf{Q} = \lambda \cdot (\mathbf{K} - \mathbf{I}) \tag{4.2}$$

where each element q_{ij} corresponds to $\lambda \cdot (1 - k_{ij})$ if $i = j$ and to $\lambda \cdot (k_{ij})$ otherwise. The parameter $\lambda = \max\{-q_{ii}\}$ is intentionally fixed to 0.2.

Moderate workload chain

Sometimes, it may happen that the workload of a CPU is not particularly intense or that it does not have operations to perform except those essential for the system. In this last case, its state is called *idle*. These two situations represent the best chances to save some energy by lowering the operating frequency to the minimum level.

The first chain we define, distinguishes a *moderate workload level*. In a model of that type, the probability to be in states WL_1 or WL_2 is greater compared to the other. This, can be intuitively described by assigning an higher transition probability to the arcs ingoing to these states as can be seen in the following figure:

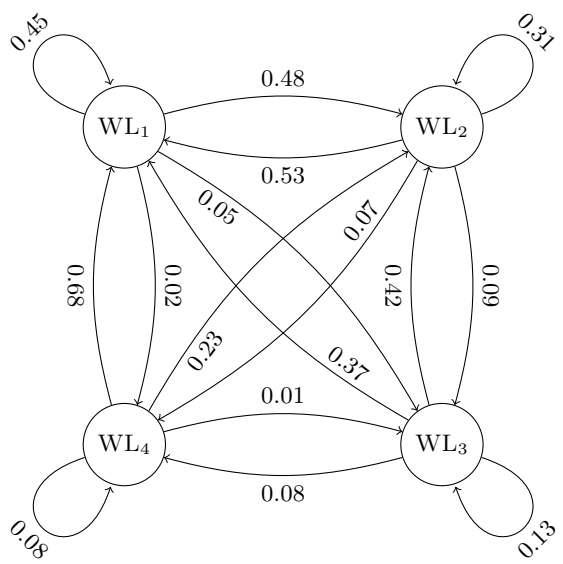

Figure 4.6: The transition probabilities of a moderate CPU *workload.*

According to what has been said in section 1.3, we organize the arcs of the model in figure 3.1 in a stochastic non-negative transition probability matrix [25, 5], $P^{(1)}$:

$$P^{(1)} = \begin{pmatrix} 0.45 & 0.48 & 0.05 & 0.02 \\ 0.53 & 0.31 & 0.09 & 0.07 \\ 0.37 & 0.42 & 0.13 & 0.08 \\ 0.68 & 0.23 & 0.01 & 0.08 \end{pmatrix}$$

Finally, using the formula (4.2), we get the corresponding infinitesimal generator $Q^{(1)}$:

$$Q^{(1)} = \begin{pmatrix} -0.110 & 0.096 & 0.010 & 0.004 \\ 0.106 & -0.138 & 0.018 & 0.014 \\ 0.074 & 0.084 & -0.174 & 0.016 \\ 0.136 & 0.046 & 0.002 & -0.184 \end{pmatrix}$$

Intense workload chain

The definition of the second chain reverses the situation. Now we want to characterize a CPU subjected to an *extreme workload level*. This occurs quite often when analyzing real systems. In general a peak workload can occur in an unforeseeable way at any moment in time.

We assign an higher transition probability to the arcs ingoing in the states WL_3 and WL_4. We follow the same process as before. The behavior of the workload is shown in the following figure:

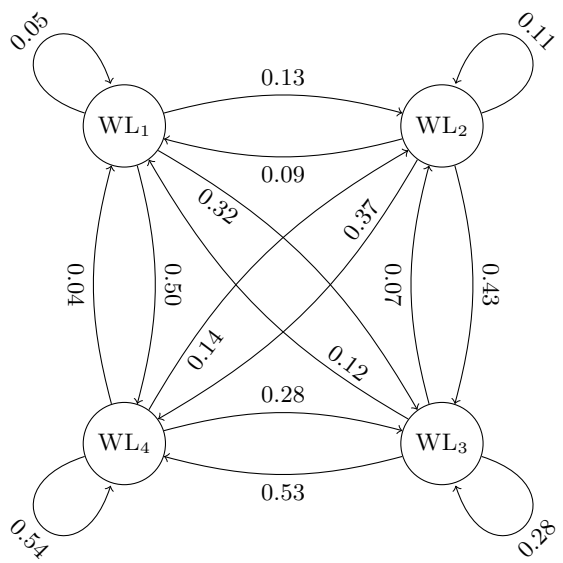

Figure 4.7: The transition probabilities of a instense CPU *workload.*

which can be translated into the corresponding transition probability matrix $P^{(2)}$:

$$P^{(2)} = \begin{pmatrix} 0.05 & 0.13 & 0.32 & 0.50 \\ 0.09 & 0.11 & 0.43 & 0.37 \\ 0.12 & 0.07 & 0.28 & 0.53 \\ 0.04 & 0.14 & 0.28 & 0.54 \end{pmatrix}$$

and finally converted into the infinitesimal generator $Q^{(2)}$ using the formula (4.2):

$$Q^{(2)} = \begin{pmatrix} -0.190 & 0.026 & 0.064 & 0.100 \\ 0.018 & -0.178 & 0.086 & 0.074 \\ 0.024 & 0.014 & -0.144 & 0.106 \\ 0.008 & 0.028 & 0.056 & -0.092 \end{pmatrix}$$

It is particularly interesting to check the success of this operation. To do this, we proceed in two ways. A first control is performed through the simulation of both the chains. The function `model_function_simulation` (A.3), has been specifically developed for this purpose. The inputs are the infinitesimal generator `Q` and the duration `d` of the simulation, while it is assumed that the initial vector is $\boldsymbol{\pi}_0 = [1, 0, 0, 0]$. The simulation results are shown in the two following figures:

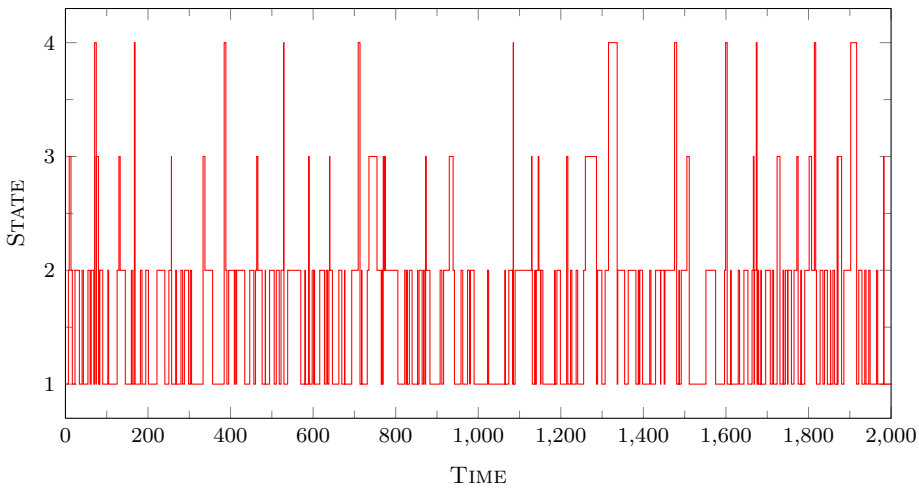

Figure 4.8: Simulation of the moderate workload chain. It is evident the behavior of the load that remains in the states WL_1 and WL_2 for most of the time.

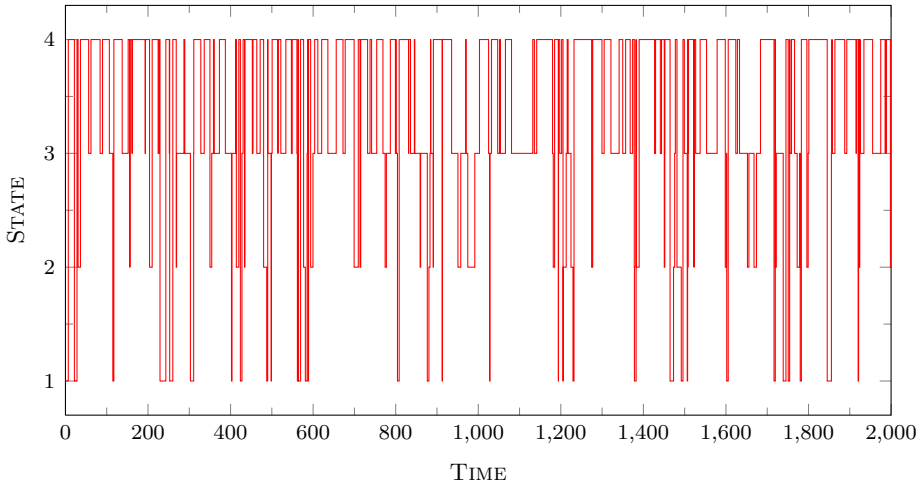

Figure 4.9: Simulation of the intense workload chain. The trace behaves in opposite ways compared to the previous figure.

As further proof, we analyze the limiting distribution of the chain. Recalling what has been said in section 1.3.3, the steady-state probability distribution may be simply obtained by solving the system of linear equations $\boldsymbol{\pi} \cdot \boldsymbol{Q} = 0$ subject to the condition $\|\boldsymbol{\pi}\|_1 = 1$. Intuitively, it is sufficient to analyze the steady-state probability vector $\boldsymbol{\pi}$. As regards the first chain, we must verify that the probabilities of the states WL_1 and WL_2 are far greater compared to the probabilities of states WL_3 and WL_4. After the analysis we obtained the steady-state probability vector of $\boldsymbol{\pi}^{(1)} = [0.4869, 0.3967, 0.0696, 0.0468]$. For the second chain, reasoning is exactly the opposite. We must verify that the probabilities of the states WL_3 and WL_4 are well above than those of the states WL_1 and WL_2. After analysis, the probability vector is $\boldsymbol{\pi}^{(2)} = [0.0705, 0.1148, 0.3000, 0.5147]$. Thus, we tested the good construction of the chains.

4.3.2 Energy aware policies for frequency regulation

We move now to define the frequency policies. What we do is construct six different frequency models, named with the letters of greek alphabet. Policies P_α, P_β and P_γ are three simply two states frequency models. We define a state in which the frequency is set to a *low level* and one in which the *high level* corresponds to a full speed of processor. Policies P_δ, P_ϵ and P_ζ are three states model, in which an intermediate frequency level is introduced. The following section presents the six models. Note that, to simplify the representation, the notation has been a little modified. If there are many different transitions from a state to any other state, we write $\{w_1, w_2, w_3\}$. The notation indicates there are three transitions with same action rate but with different action type.

Two levels frequency regulatory policies

The policies P_α, P_β and P_γ, are constituted by two frequency levels. What we do is adjust the balancing of the model, by grouping together in different ways the activities on the

states. The first policy is shown in the following figure:

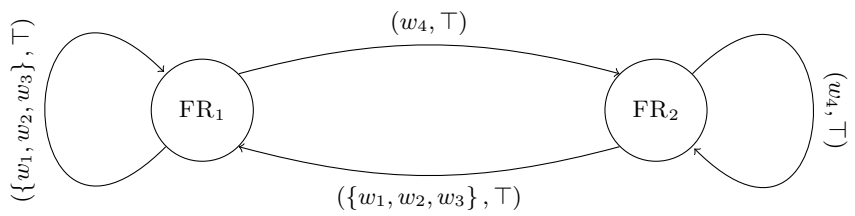

Figure 4.10: Graphical representation of P_α policy.

From state FR_1, the action types w_1, w_2 or w_3 keeps the model in state FR_1. An action type w_4 enables the jump to the state FR_2. On the opposite side, from state FR_2, the w_4 action keeps the model in the same state. The model is enabled to jump to state FR_1 with the other actions. Thus, aggregation is on the state FR_1 because this gathers most of the actions. In Pepa language the policy correspond to:

$$\begin{array}{lcllllllll} FR_1 & \stackrel{def}{=} & (w_1, \top).FR_1 & + & (w_2, \top).FR_1 & + & (w_3, \top).FR_1 & + & (w_4, \top).FR_2 \\ FR_2 & \stackrel{def}{=} & (w_1, \top).FR_1 & + & (w_2, \top).FR_1 & + & (w_3, \top).FR_1 & + & (w_4, \top).FR_2 \end{array}$$

The second policy is built providing a different activities aggregation. From state FR_1, the action type w_1 keeps the model in the same state. Action types w_2, w_3 and w_4 enables the jump to the state FR_2. On the opposite side, from state FR_2 the action types w_2, w_3 and w_4 keeps the model in the same state. Finally, the model is enabled to jump to the state FR_1 with the action type w_1. This behavior is shown in the following figure:

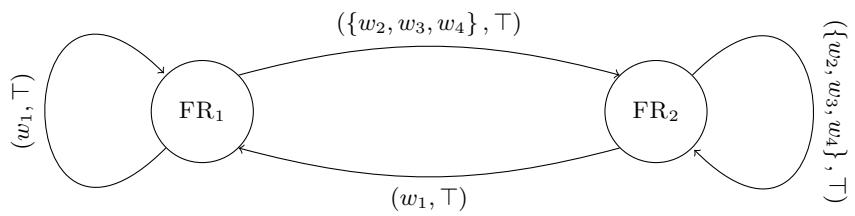

Figure 4.11: Graphical representation of P_β policy.

Thus, aggregation is on the state FR_2. In Pepa language the policy correspond to:

$$
\begin{array}{lcllllllll}
FR_1 & \stackrel{def}{=} & (w_1, \top).FR_1 & + & (w_2, \top).FR_2 & + & (w_3, \top).FR_2 & + & (w_4, \top).FR_2 \\
FR_2 & \stackrel{def}{=} & (w_1, \top).FR_1 & + & (w_2, \top).FR_2 & + & (w_3, \top).FR_2 & + & (w_4, \top).FR_2
\end{array}
$$

In the third policy, we equivalently distribute activities between the states. This is shown in the following figure:

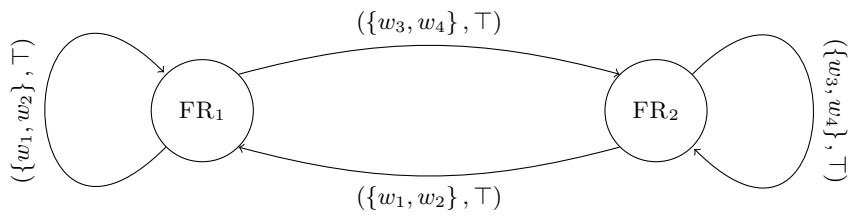

Figure 4.12: Graphical representation of P_γ policy.

From state FR_1, the action types w_1 and w_2 keeps the model in state FR_1. The action types w_3 and w_4 enables the jump to the state FR_2. On the opposite side, from state FR_2 the action types w_3 and w_4 keeps the model in the same state. The model is enabled to jump in state FR_1 with the other actions. Thus, aggregation is balanced among the two states. In PEPA language the policy correspond to:

$$
\begin{array}{lcllllllll}
FR_1 & \stackrel{def}{=} & (w_1, \top).FR_1 & + & (w_2, \top).FR_1 & + & (w_3, \top).FR_2 & + & (w_4, \top).FR_2 \\
FR_2 & \stackrel{def}{=} & (w_1, \top).FR_1 & + & (w_2, \top).FR_1 & + & (w_3, \top).FR_2 & + & (w_4, \top).FR_2
\end{array}
$$

Three levels frequency regulatory policies

The policies P_δ, P_ϵ and P_ζ, are constituted by three frequency levels. In these models, it is evident the importance of the transition strategy. As explained in section 3.2.2, we choose to build a frequency model that attempts to maintain an acceptable level of throughput.

This is obtained by developing a policy that force the model to reach immediately the high speed level and then gradually move down to the slow speed level by passing for the intermediate state. The first three states policy is shown in the following figure:

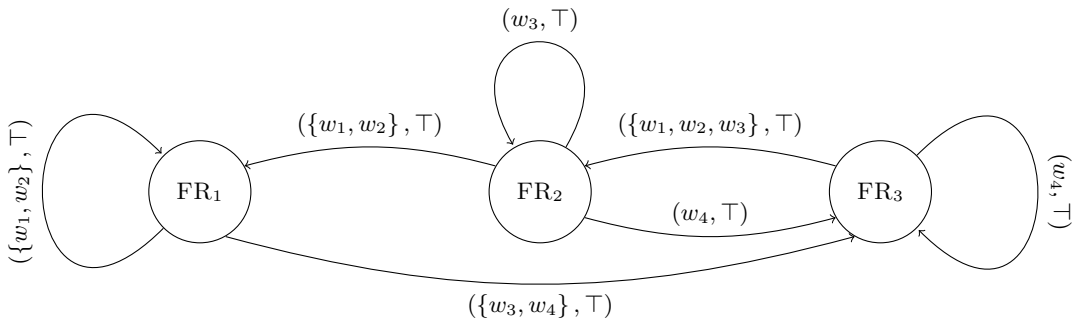

Figure 4.13: Graphical representation of P_δ policy.

From state FR_1, the action types w_1 and w_2 keeps the model in state FR_1. The action types w_3 and w_4 enables the jump directly to the state FR_3. From state FR_2 the w_3 action, keeps the model in the same state. The model is enabled to jump in state FR_1 with the action types w_1 and w_2 and in state FR_3 with the action type w_4. From state FR_3, the action type w_4 keeps the model in state FR_1. The action types w_2, w_3 and w_4 enables the jump to the state FR_2. Thus, in this policy, the aggregation is on FR_1. In PEPA language, this behavior correspond to:

$$
\begin{array}{lcllllllll}
FR_1 & \stackrel{def}{=} & (w_1, \top).FR_1 & + & (w_2, \top).FR_1 & + & (w_3, \top).FR_2 & + & (w_4, \top).FR_3 \\
FR_2 & \stackrel{def}{=} & (w_1, \top).FR_1 & + & (w_2, \top).FR_1 & + & (w_3, \top).FR_2 & + & (w_4, \top).FR_3 \\
FR_3 & \stackrel{def}{=} & (w_1, \top).FR_2 & + & (w_2, \top).FR_2 & + & (w_3, \top).FR_2 & + & (w_4, \top).FR_3
\end{array}
$$

The second policy is built providing a different activities aggregation. From state FR_1, the action type w_1 keeps the model in state FR_1. The action types w_2, w_3 and w_4 enables the jump directly to the state FR_3. From state FR_2 the action type w_2 keeps the model in the same state. The model is enabled to jump in state FR_1 with the action type w_1 and in state FR_3 with the action types w_3 and w_4. From state FR_3, the action types w_3 and w_4 keeps the model in state FR_3. Finally, the action types w_1 and w_2 enables the jump to the state FR_2. This behavior is shown in the following figure:

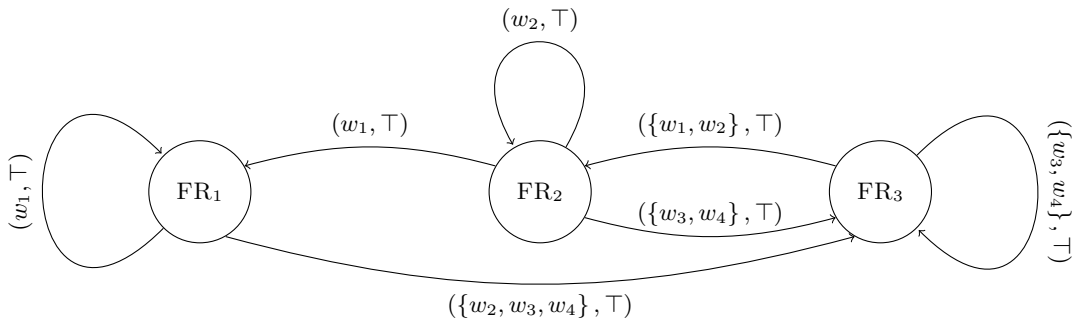

Figure 4.14: Graphical representation of P_ϵ policy.

Thus, aggregation is on the state FR_3. In PEPA language the policy correspond to:

$$\begin{array}{lcllllllll}
FR_1 & \stackrel{def}{=} & (w_1, \top).FR_1 & + & (w_2, \top).FR_3 & + & (w_3, \top).FR_3 & + & (w_4, \top).FR_3 \\
FR_2 & \stackrel{def}{=} & (w_1, \top).FR_1 & + & (w_2, \top).FR_2 & + & (w_3, \top).FR_3 & + & (w_4, \top).FR_3 \\
FR_3 & \stackrel{def}{=} & (w_1, \top).FR_2 & + & (w_2, \top).FR_2 & + & (w_3, \top).FR_3 & + & (w_4, \top).FR_3
\end{array}$$

In the last policy we try to equivalently distribute activities among the states. This is shown in the following figure:

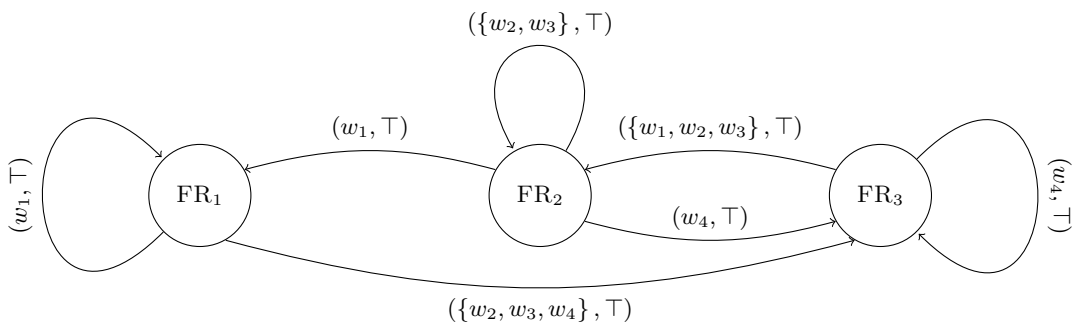

Figure 4.15: Graphical representation of P_ζ policy.

From state FR_1, the action type w_1 keeps the model in state FR_1. The action types w_2, w_3 and w_4 enables the jump directly to the state FR_3. From state FR_2 the action types w_2 and w_3 keeps the model in the same state. The model is enabled to jump in state FR_1 with the action type w_1 and in state FR_3 with the action type w_4. From state FR_3, the action type w_4 keeps the model in state FR_1. The other activities enable the jump to the state FR_2. Thus, aggregation is on the state FR_2. In PEPA language the policy correspond to:

$$
\begin{array}{lllllllll}
FR_1 & \stackrel{def}{=} & (w_1, \top).FR_1 & + & (w_2, \top).FR_3 & + & (w_3, \top).FR_3 & + & (w_4, \top).FR_3 \\
FR_2 & \stackrel{def}{=} & (w_1, \top).FR_1 & + & (w_2, \top).FR_2 & + & (w_3, \top).FR_2 & + & (w_4, \top).FR_3 \\
FR_3 & \stackrel{def}{=} & (w_1, \top).FR_2 & + & (w_2, \top).FR_2 & + & (w_3, \top).FR_2 & + & (w_4, \top).FR_3
\end{array}
$$

In order to derive the performance measure, we must assign a frequency value to each states of the model. This has already been mentioned in section 3.3. The frequency values of the workload model are fixed at the start. They correspond to the levels 500, 1000, 1500 and 2000 respectively for the state WL_1, WL_2, WL_3, and WL_4. In order to maintain as possible high the level throughput, to each state of the frequency model must be assigned the maximum value among all of possible states of the WL model that are mapped into it. An example is provided by the following figure:

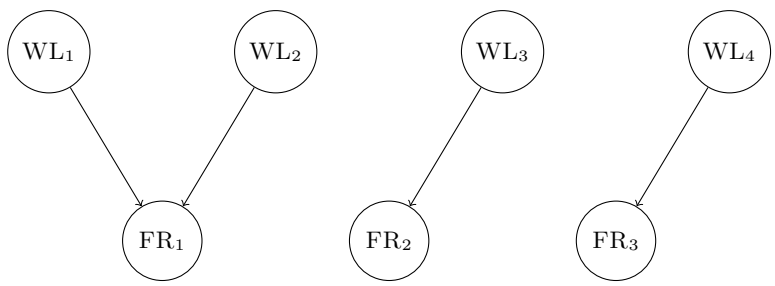

Figure 4.16: Example of frequency level association. In this case the state FR_1 gathers the states WL_1 and WL_2. Therefore, its frequency level must be the same of WL_2.

We formally describe this function. We can surely say that it is a surjective but not injective. Let $\omega(w_i) : \mathcal{W} \to \mathbb{R}^+$, $\phi(f_i) : \mathcal{F} \to \mathbb{R}^+$ where $WL_i, i = 1 \ldots n \in \mathcal{W}$ and $FR_i, i = 1 \ldots n \in \mathcal{F}$, two function to assign a real positive number (the frequency level) to states of WL and FR models respectively and let $\theta(w_i) : \mathcal{W} \to \mathcal{F}$ the function that maps the states of workload model in the states of frequency model. Then $\phi(f_i) = \max\{\omega(w_i)\}\ \forall w_i \in \theta^{-1}(f_i)$. According to this formalization, the frequency values to assign to each state for each policy are given in the following table.

Policy	States						
	WL_1	WL_2	WL_3	WL_4	FR_1	FR_2	FR_3
P_α	500	1000	1500	2000	1500	2000	
P_β	500	1000	1500	2000	500	2000	
P_γ	500	1000	1500	2000	1000	2000	
P_δ	500	1000	1500	2000	1000	1500	2000
P_ϵ	500	1000	1500	2000	500	1000	2000
P_ζ	500	1000	1500	2000	500	1500	2000

Table 4.1: Frequency levels assigned to the model states for each regulatory policy.

4.4 Results

We present now the results obtained with different parameterizations of the model. We, first, show the behavior of the CPU subjected to a moderate workload and subsequently to a intense workload. In the end we evaluate through the cost function, the global behavior of the model.

4.4.1 Moderate workload

The first two graphs show the behavior of the performance indexes of the model for different values of the parameter r. Recalling that $r \in [10^{-2}, 10^{-1}, 10^{0}, 10^{1}, 10^{2}, 10^{3}]$, the results, under a moderate level of workload, are:

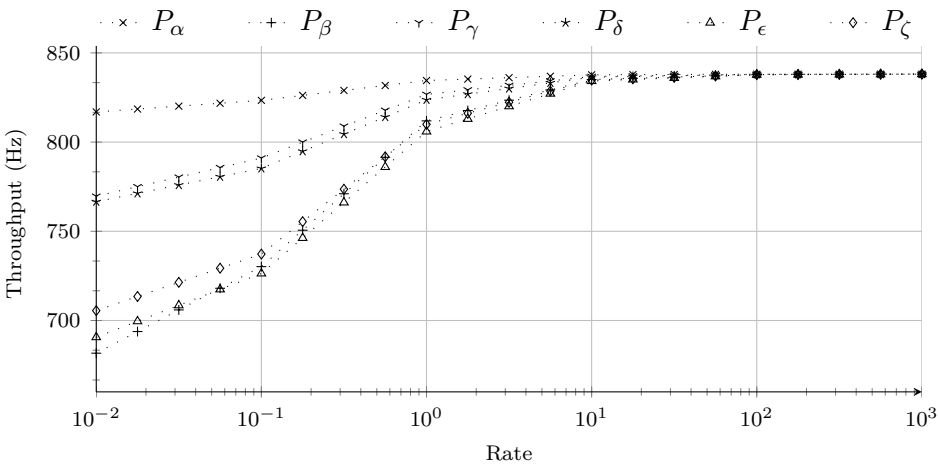

Figure 4.17: The change in the value of the throughput. It's clearly noted that, more this value increases, more throughput approaches to the maximum theoretical value ($838.12Hz$*, obtained via a processor model that always works at maximum speed).*

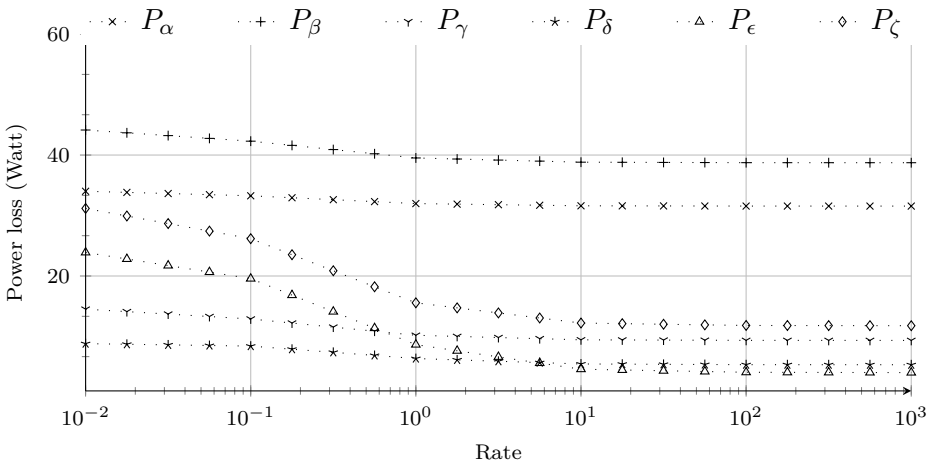

Figure 4.18: The change in the value of the power loss. It is calculated as the difference between the power actually consumed by the CPU *and that consumed in a ideal frequency level. The maximum theoretical power loss is* $86.6644Watt$*.*

Throughput

We show now the bar graph that displays the variation of the level of throughput. We include only the values obtained with parameter $r = [10^{-2}, 10^{-1}, 10^{2}, 10^{3}]$. It should be noted that the value $r = 10^3$ is the best achievable result.

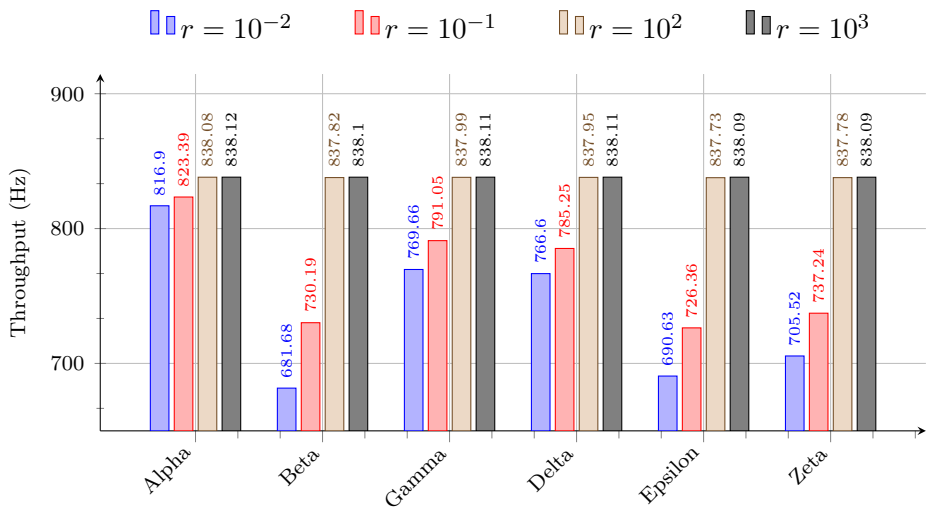

Figure 4.19: Throughput level at moderate workload. From the graph it is evident that the policy P_α, is less sensitive to variations of the parameter r, unlike P_β, P_ϵ and P_ζ policy.

Power loss

We now show the bar graph that showing the variation of the level of power loss. We also show the percentage of improvement for this measure compared to the level of power loss maximum equal to $86.6644 Watt$.

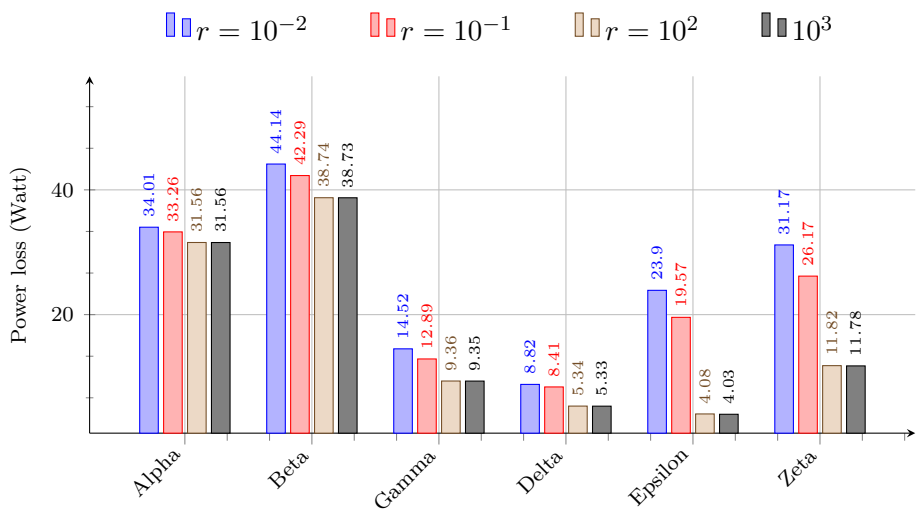

Figure 4.20: Power loss at moderate workload. The graph shows that the increase of the frequency levels results in the decrease of power loss.

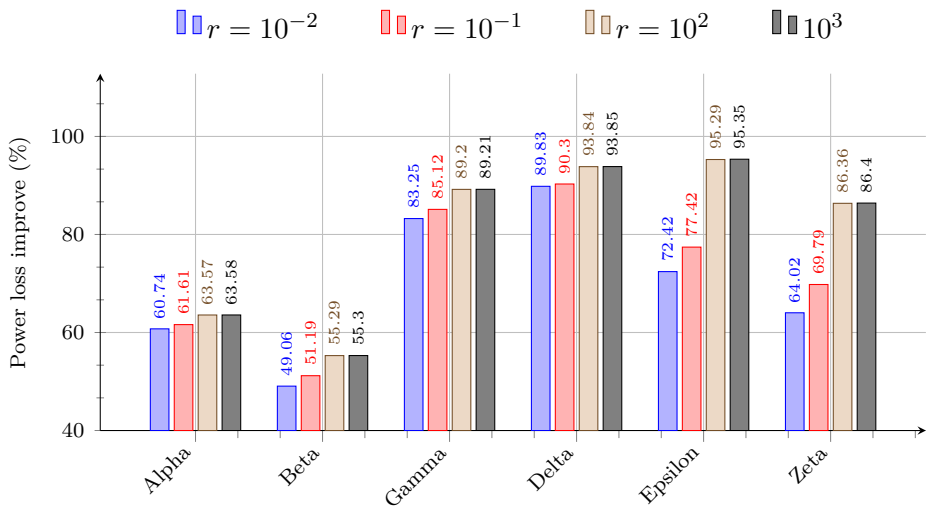

Figure 4.21: Power loss variations at moderate workload. The graph shows that the best performance in terms of power loads are achieved by the models P_δ and P_ϵ.

Cost function

We are now ready to see a comprehensive assessment of the energy performance. To do this we use the cost function $C(\alpha, \beta)$ defined in section 3.3.3. The parameters α and β are respectively set to 0.3 and 0.7

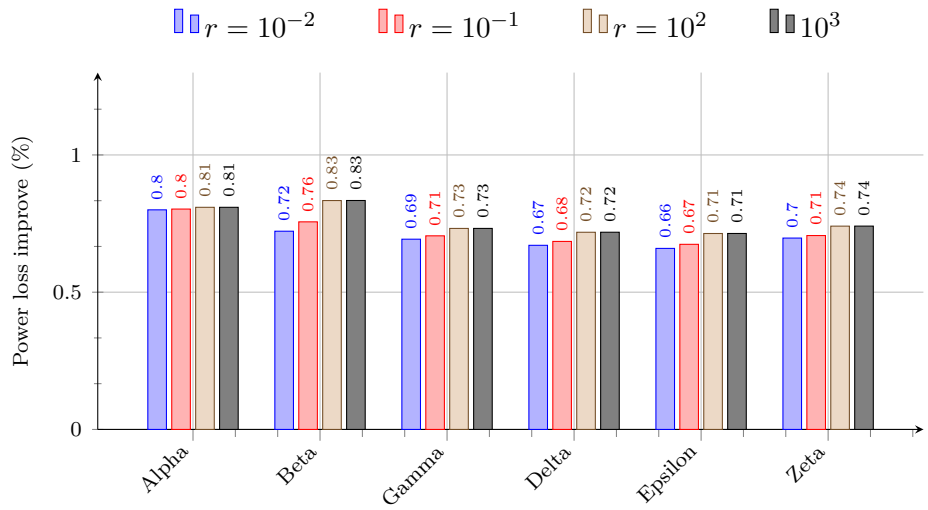

Figure 4.22: Cost function for the moderate workload. The graph shows that the best frequency regulatory policy for a workload of this type is P_ϵ (because we are looking for the minimum value of the function).

4.4.2 Intense workload

The first two graphs show the behavior of the performance indexes of the model for different values of the parameter r. Recalling that $r \in [10^{-2}, 10^{-1}, 10^{0}, 10^{1}, 10^{2}, 10^{3}]$, the results, under a intense level of workload, are:

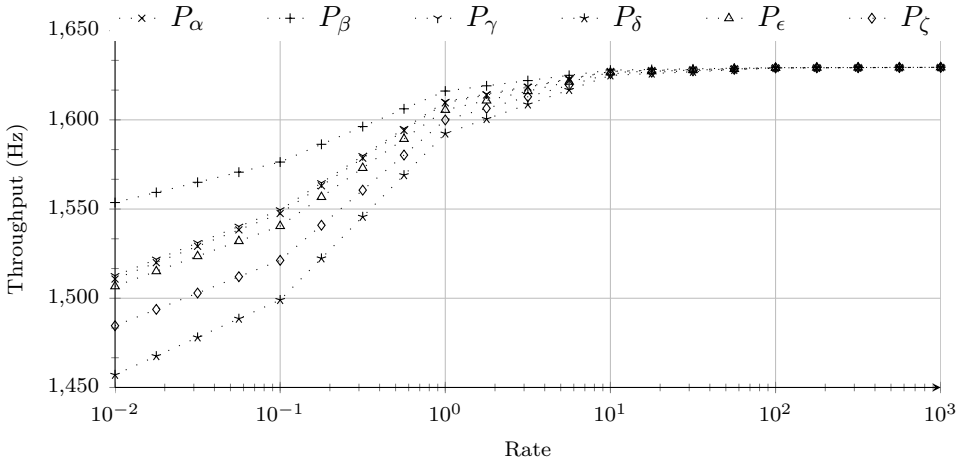

Figure 4.23: The change in the value of the throughput. It's clearly noted that, more this value increases, more throughput approaches to the maximum theoretical value ($1629.45Hz$*, obtained via a processor model that always works at maximum speed).*

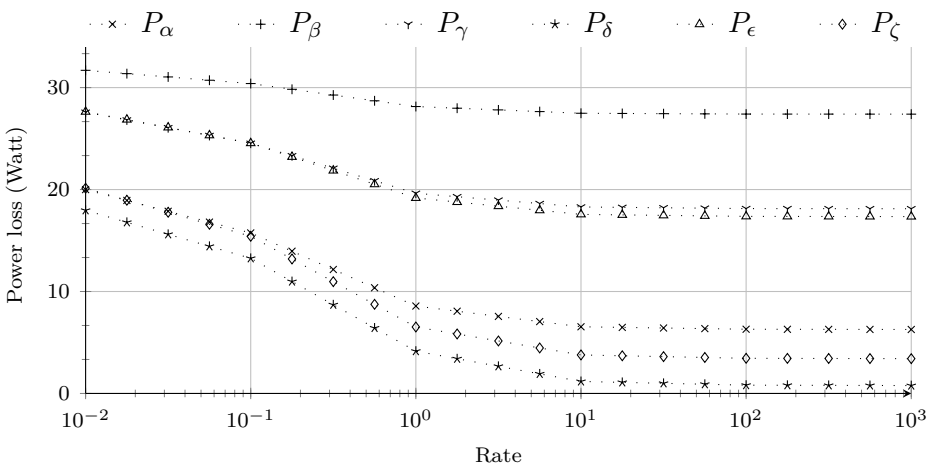

Figure 4.24: The change in the value of the power loss. It is calculated as the difference between the power actually consumed by the CPU *and that consumed in a ideal frequency level. The maximum theoretical power loss is* $34.3304Watt$.

Throughput

We show now the bar graph that displays the variation of the level of throughput. We include only the values obtained with parameter $r = [10^{-2}, 10^{-1}, 10^{2}, 10^{3}]$. It should be noted that the value $r = 10^3$ is the best achievable result.

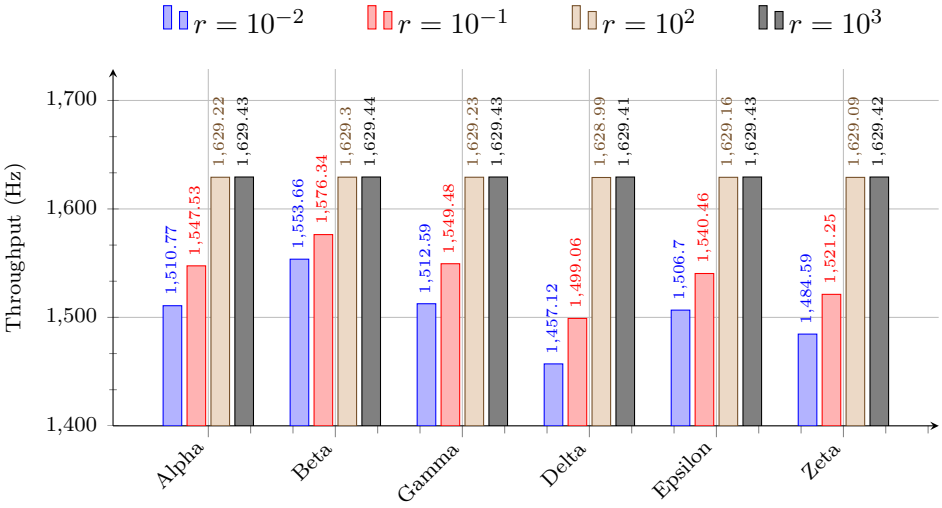

Figure 4.25: Throughput level at intense workload. From the graph seems that all policies follow an almost identical trend.

Power loss

We now show the bar graph that showing the variation of the level of power loss. We also show the percentage of improvement for this measure compared to the level of power loss maximum equal to $34.3304 Watt$.

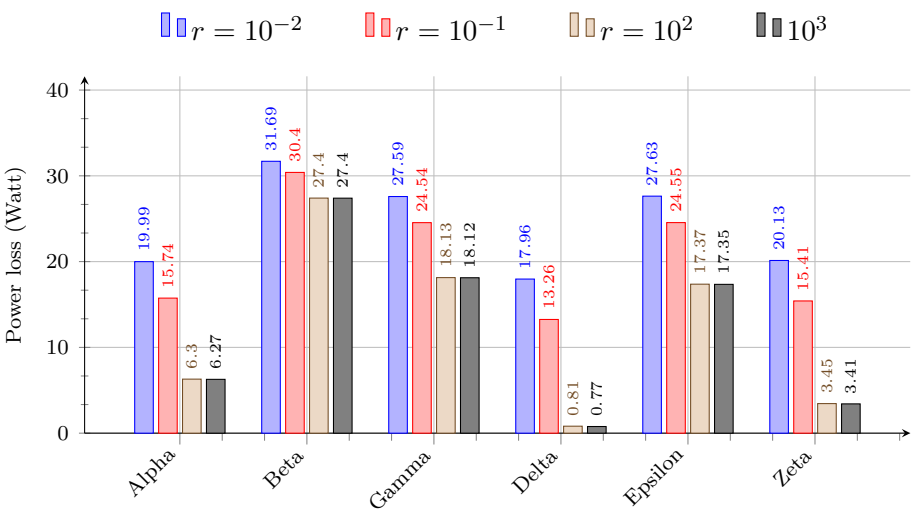

Figure 4.26: Power loss at intense workload. The graph shows that the increase of the frequency levels results in the decrease of power loss.

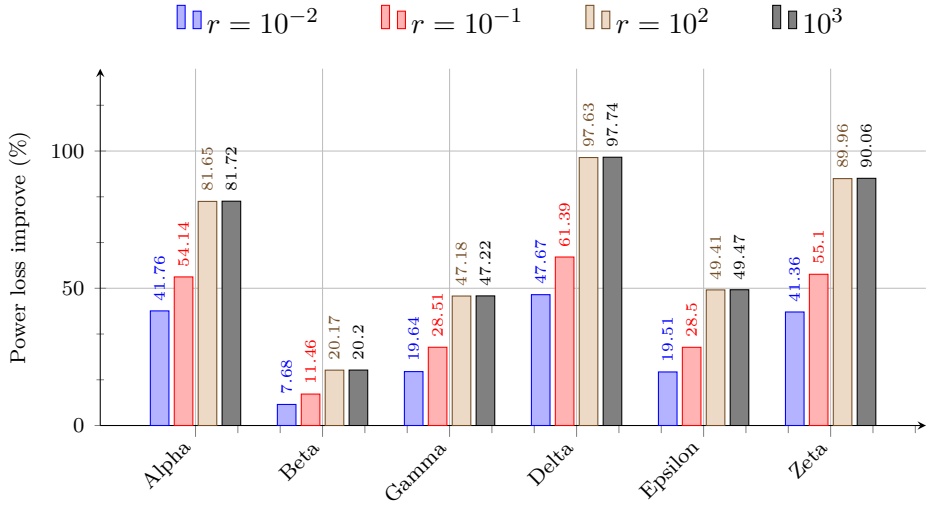

Figure 4.27: Power loss variations at intense workload. The graph shows that the best performance in terms of power loads are achieved by the models P_δ and P_ζ.

Cost function

We are now ready to see a comprehensive assessment of the energy performance. To do this we use the cost function $C(\alpha, \beta)$ defined in section 3.3.3. The parameters α and β are respectively set to 0.3 and 0.7

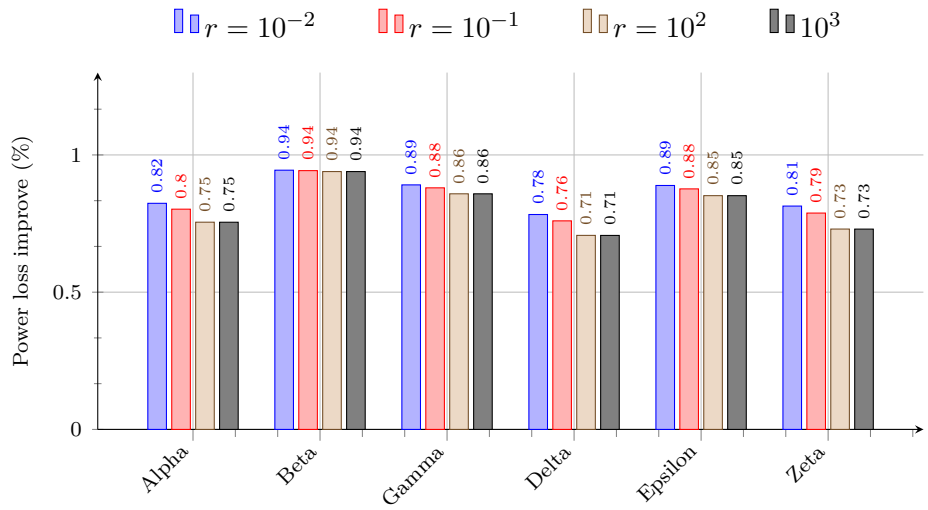

Figure 4.28: Cost function for the intense workload. The graph shows that the best frequency regulatory policy for a workload of this type is P_δ (because we are looking for the minimum value of the function).

4.5 Conclusion

In this chapter, we presented the algorithm for the automatic formalization in process algebra PEPA and for the subsequent collection of measures. We explained in the detail all its components and how it works. In particular, the inputs are constituted by two different continuous-time Markov chains and by a set of frequency regulatory policies. The chains, represent the behavior of the workload of the CPU subjected to a both moderate and intense workload. The policies consist of two and three frequency levels with different states aggregations.

Starting from this parameterization, we performed several tests in order to evaluate the behavior of the model. The obtained results show how it is possible to quickly derive the performance measures in a purely algorithmic way by the analysis of the underlying Markov chain of the model.

Chapter 5

Case study on Google datacenter workload data

5.1 Introduction

In the previous chapter, we presented the algorithm used to automate the model analysis and the performance measures derivation process. We also performed some tests starting from an analytical parameterization. What we do in this chapter is to study the behavior of the model when it is parameterized starting from a real CPU workload trace.

The model has been applied to evaluate the performance of the same set of frequency control strategies on the traces provided by the *GoogleClusterData* project [24], that provides detailed information about the workload of a cluster of about 11,000 machines. In order to parameterize the model, traces have been fitted into Markovian process. The performance measures has been evaluated using the algorithm described above.

The chapter is structured as follows: in section 5.2 we will present the data of the *GoogleCluster* project by defining the semantic, the format and the schema of the information it provides (5.2.1) and the procedure to extract the workload of each machine (5.2.2). In sections 5.3 and 5.4, we will show the parametrization of the model. Finally, the results will be presented.

5.2 The GoogleClusterData project

In this section, we describe the information released by *Google* through the *GoogleClusterData* project [24]. A *Google* cluster is defined as a collection of machines connected by an high speed network and contained in a *rack*. A subset of these machines is called *cell*, a set of physical machines that are operated as a single unit and shares a cluster manager that allocates to each machines the work to execute. The user requests reaches the cell as *jobs*, each of which is comprised of one or more *tasks*. A task usually represent a program and, as such, it is composed by multiple processes.

Data come from two distinct sources. The first is the cluster manager, which allocates the requests from users to the machines of the cell. The second are the machines themselves. However, our interest is mainly focused on the machines and their individual workload. It is known that the machines configuration is not homogenous. Unfortunately, due to the obfuscation of some data, is not possible to know the exact machine configuration and its architecture.

Anonymization has been implemented in several ways. In particular, most fields have been randomly hashed, resource sizes have been scaled and certain values have been mapped into a sorted series. To identify the different machines, a 64bits unique and never reused identifier is assigned to them. Thus, the trace identifies a user associated with each job and the machine to which it is assigned.

Starting from the described architecture, Google, in May 2011 has traced about 25 million tasks distributed between 11,000 heterogeneous machines. The traces are given in the well-known CSV format and contains the description of 29 days of workload in one cell. During the monitoring, more than 10 metrics are collected. Metrics ranging from the CPU usage to the memory usage until the cycles per instruction (CPI). All this data are tabulated and grouped according to common characteristics.

5.2.1 Data tables

In total, the *GoogleClusterProject* offers about 40GB of information regarding the behavior of a cluster. The informations are grouped together according the subject of analysis. To share the collected data in a clear and tidy way, *Google* proceeded to divide them on different tables:

Machines tables
Tables *Machine events* and *Machine attribute*, contain respectively all the events that affect a machine and the information on it. An event regards of all the operations of addition and removal of a machine from the cluster and the updating of available resources. The machine attributes include some information on the configuration of a particular machine. For example, the kernel version, the clock speed and the IP address are given. Unfortunately, the information regarding the configuration are hashed preventing to recognize the system architecture.

Tasks tables
Information concerning tasks are divided between tables *Task events*, *Task constraints*, and *Task usage*. The first table lists the information and events of a specific task. In particular, it includes information about the job and the machine to which it is assigned, the position inside the job, the priority of execution and also the maximum amount of resources that can be used (in terms of CPU, RAM and disk space).

The second table contains information about the restrictions on the execution of a task (usually it may be decided a particular task can not be run from a machine with a specific

configuration). Finally, the last table, lists all the statistics related to the execution of a task and its resource consumption.

Jobs tables

Informations on the jobs are collected in a single table. The *Job events* table contains all the information on a specific job. The following diagram shows a simplified version of the jobs and tasks lifecycle:

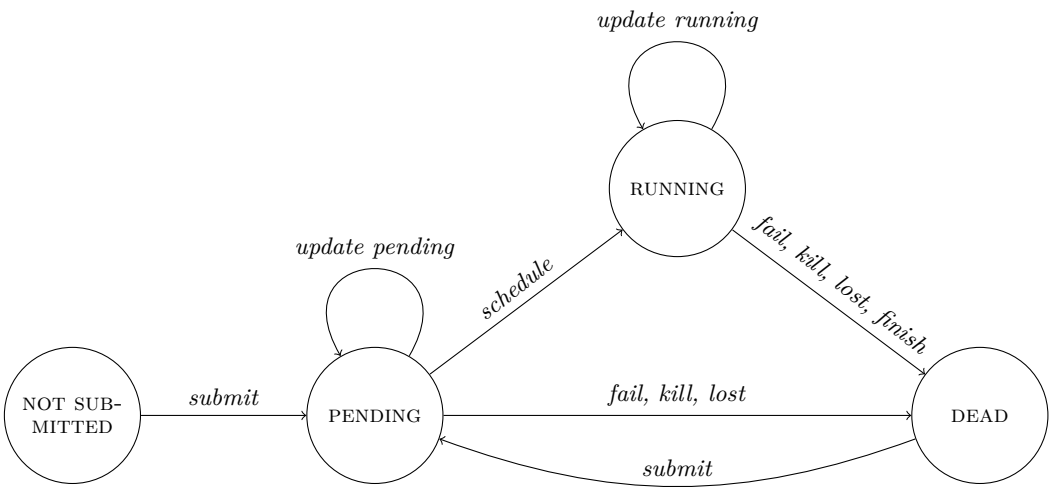

Figure 5.1: Jobs and tasks lifecycle and event types.

The table contains only the informations about the jobs that are in the RUNNING or PENDING state. There is also a field that collects the type of event which has occurred, one which indicates the identifier of the job and one for the scheduling priority.

This is only a brief description of the data provided. A more accurate description can be found in [24]. However, table *Task usage* contains the informations essential for our analysis, so we proceed to give an accurate definition of it:

	Field content	Field type	Mandatory
1	Start time	INTEGER	✓
2	End time	INTEGER	✓
3	Job ID	INTEGER	✓
4	Task index	INTEGER	✓
5	Machine ID	FLOAT	✓
6	CPU rate	FLOAT	
7	Canonical memory usage	FLOAT	
8	Assigned memory usage	FLOAT	
9	Unmapped page cache	FLOAT	
10	Total page cache	FLOAT	
11	Maximum memory usage	FLOAT	
12	Disk I/O time	FLOAT	

	Field content	Field type	Mandatory
13	Local disk space usage	FLOAT	
14	Maximum CPU rate	FLOAT	
15	Maximum disk I/O time	FLOAT	
16	Cycles per instruction	FLOAT	
17	Memory accesses per instruction	FLOAT	
18	Sample portion	FLOAT	
19	Aggregation type	BOOLEAN	

Table 5.1: Table Task usage fields description.

From the description of the fields, may be immediately understood the importance of the information it contains. There are, in fact the measurements concerning CPU utilization of each task, as well as the identifier of the machine to which it is assigned and the measurement time interval. As regards the measurement interval, the fields `Start time` and `End time` identifying respectively the beginning and the end of the measurement window. These times are expressed in microseconds staring form 600 seconds after the start of the trace session and recorded in 64bits. In the supplied data, all times and timestamps are defined in this way. The following figure helps us to clarify the foregoing:

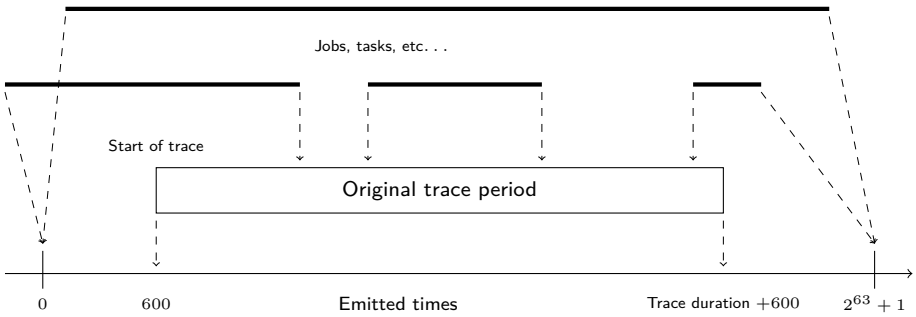

Figure 5.2: Mapping between the original times and trace times in the GoogleClusterData project.

where the special values 0 and $2^{63} - 1$, represents respectively an event occurred before the start of the trace session and one that ends after the end of the observation period. These considerations about the format of the times are essential as, the `Cpu rate` (and all other collected indexes), is reported as the average of the CPU rates within a measuring window. This window is typically 300 seconds and a measurement is made every second. In some cases may occur, due to the excessive workload, that the measurement rate is not respected. The sample rate field, defined as the ratio of the number of observations planned and executed, keeps track of these situations.

The described fields, however, represent the CPU usage of each task on a set of machines in a time interval and they are not directly usable to derive the total workload of a generic machine. In order to obtain a complete workload trace of each machine, the data files must be, first of all, cleaned.

5.2.2 Workload extrapolation

As explained, the data which are necessary to extract the workload of a single machine are included in the table *Task usage*. This collects statistics on the resources consumption, of the different machines in the cluster, by the tasks. For the reason that the informations are divided by task and provided all together, they are not directly usable to extract the workload of a specific machine but must be processed.

In order to perform the data extrapolation operation, we developed the bash script `csv-cleaner.sh` (B.1) and the two java classes `java-splitter.java` (B.3) and `java-sampler.java` (B.3). The script, takes as input, the identifier of a machine belonging to the cluster. The procedure is quite simple and is done in three steps. The first phase consists in the cleaning and ordering of data. As described, the *Task usage* table contains many data, most of which is unnecessary for our purposes. In order to reduce the amount of data to process, all unnecessary informations are deleted. After being extracted, all the unnecessary columns are deleted from the files. Subsequently, these, are ordered by moving to the first position, the column containing the machine identifiers. Finally, the lines are sorted in ascending order according to this identifier. In the second part of the script, the extraction of informations belonging to the same machine is performed. The class `java-splitter.java` executes this operation. The procedure reads the input file line by line and transfers the informations on new files whose name is equal to the machine identifier. The last part of the script is responsible of the computation of the workload. This operation is performed by the class `java-sampler.java` that receives in input the file corresponding to the desired machine. The procedure reads the file line by line and sums all CPU usages which are in the same time interval. An example is in the following figure:

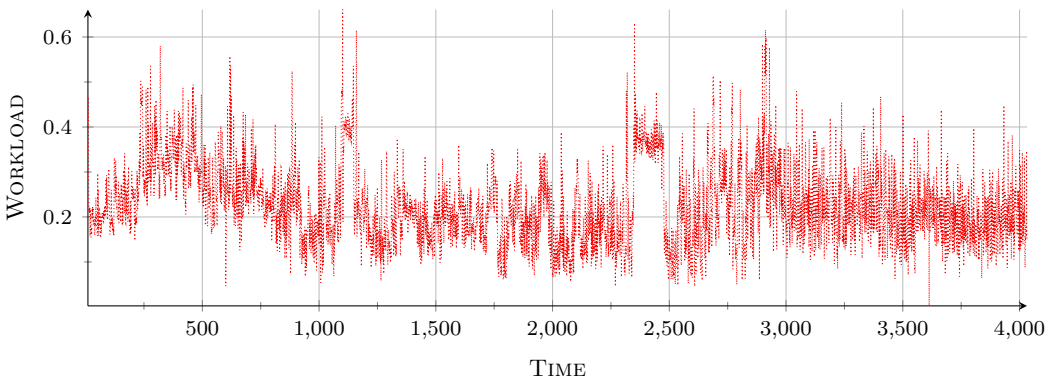

Figure 5.3: Workload of the Google machine 1404437.

5.3 Maximum Likelihood Estimation for Markov chains

Our problem is to estimate a Markov chain that gives the best possible representation of a real workload. To do this we use a method known as maximum likelihood estimation (MLE) that, starting from sequence of observed states (different frequency levels) estimates the transition probabilities matrix of a Markov chain.

Suppose [7], to observe a sequence of values $\bar{x} = x_1, x_2, x_3, \ldots, x_n$ assumed by the random variable X. What we want to do, is to estimate the transition probability matrix P of a Markov chain M with k states from the sequence $\bar{x}$. Intuitively, the values assumed by the variable X correspond to the states of the chain and the transition probability to the probability to observe a determinate sequence. Thus, we stimate the elements p_{ij} of the matrix. They correspond to $p_{ij} = Pr\{X_{t_{u+1}} = j \,|\, X_{t_u} = i\}$. The probability to observe the sequence $\bar{x}$ comes from the chain rule [25], so:

$$\begin{aligned} Pr\{X = \bar{x}\} = Pr\{X_n = x_n, \,|\, X_{n-1} = x_{n-1}, \ldots, X_1 = x_1\} \cdot \\ Pr\{X_{n-1} = x_{n-1}, \,|\, X_{n-2} = x_{n-2}, \ldots, X_1 = x_1\} \cdot \ldots \cdot Pr\{X_1 = x_1\} \end{aligned} \quad (5.1)$$

This equation can be easily simplified by applying the Markov property:

$$\begin{aligned} Pr\{X = \bar{x}\} = Pr\{X_n = x_n, \,|\, X_{n-1} = x_{n-1}\} \cdot \\ Pr\{X_{n-1} = x_{n-1}, \,|\, X_{n-2} = x_{n-2}\} \cdot \ldots \cdot Pr\{X_1 = x_1\} \end{aligned} \quad (5.2)$$

resulting in:

$$Pr\{X = \bar{x}\} = Pr\{X_1 = x_1\} \prod_{i=2}^{n} Pr\{X_i = x_i, \,|\, X_{i-1} = x_{i-1}\} \quad (5.3)$$

We rewrite equation (5.3) in terms of the transition probabilities, to get the likelihood of the transition matrix P:

$$L(P) = Pr\{X_1 = x_1\} \prod_{i=2}^{n} P_{x_{i-1}x_i} \quad (5.4)$$

Let n_{ij} be the number of times x_i is followed by x_j in $\bar{x}$. We can rewrite the previous equation, thus:

$$L(P) = Pr\{X_1 = x_1\} \prod_{i=1}^{k} \prod_{j=1}^{k} p_{ij}^{n_{ij}} \tag{5.5}$$

where k is the number of the states. At this point we have a likelihood function $L(P)$, we must maximize. Before calculating the derivative, in order to simplify, we pass to the logarithm:

$$\mathcal{L}(P) = \log L(P) = \log Pr\{X_1 = x_1\} + \sum_{i=1}^{k} \sum_{j=1}^{k} n_{ij} \cdot \log p_{ij} \tag{5.6}$$

Taking the derivative and setting it equal to zero, under the conditions that $\sum_{\forall j} p_{ij} = 1$, we may conclude that:

$$p_{ij} = \frac{n_{ij}}{\sum_{j=1}^{m} n_{ij}} \tag{5.7}$$

5.4 Fitting algorithm

What has been formally explained in previous section, is performed by the MATLAB function `model_function_fitting` (A.2), appositely developed to this purpose. The inputs consist of a `trace` corresponding to a column vector and containing the value of the CPU workload in each measurement interval (5min), the number `s` of the WL model states, a `sample`, which indicates the number of elements used to improve the correlation and `lambda` used, as explained in section 4.3.1, to build the infinitesimal generator from the transition probability matrix. The fitting procedure covers many stages. The first is a simple initialization stage in which we allocate some variables. The following, consist in the subdivision of the trace. This stage is very important because we assign a level to each detected value. What we do, is to take the minimum and the maximum value of trace and split this space in $n + 1$ zones. Each trace value, which is inside to one of these zone is replaced with a number $i = 0 \ldots n$. For example, if a value is inside the second zone, it will be replaced with 1. The result is in the column vector `ft`. An example of this construction is in the following figure:

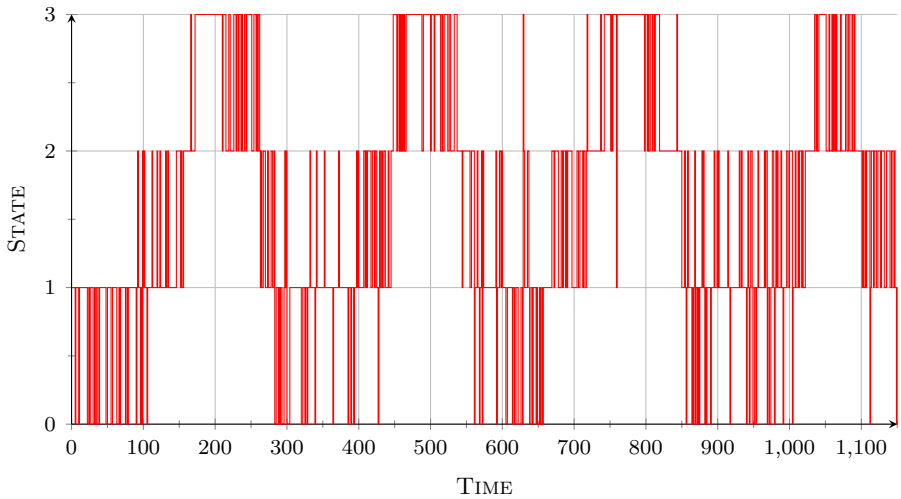

Figure 5.4: Example of the level assignment. Each trace value which is inside to one of the zone is replaced with the number of the zone itself. In this way we pass from a workload value to a frequency level.

The next step aims to increase the correlation. The above operation has resulted in a loss of informations that are reintroduced. At this stage, we chose a window of `k` elements, then we iterate through the `ft` vector. For each element, we sum the previous k elements of it. Then we multiply this value for the number of state and we sum the current state. An example of this second step is:

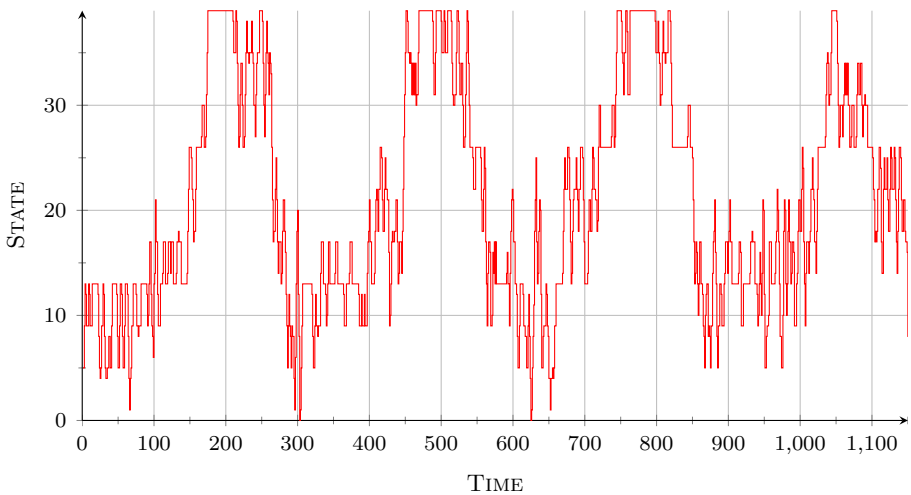

Figure 5.5: The procedure to recover the loss correlation leads to an increase in the number of states.

This procedure is organized in the following equation. Let i the index of current state and s the number of states:

$$ST(i) = \left[s \cdot \sum_{k=i-n-1}^{i-n} FT(k) \right] + FT(i) \tag{5.8}$$

The vector `st` is the sequence of states $\bar{x} = x_1, x_2, x_3, \ldots, x_n$ of which we want to estimate the probability using the definition given in section 5.3. Applying what has been said, it is easy to obtain the matrix of transition probabilities P. In the construction of probability matrix sometimes may happen that there is no transition from a state because this is never reached. In this way the chain is not ergodic. To solve this important problem we consider only the ergodic subcomponents of the matrix. The never reached states, will be eliminated from the chain. This operation opens a new problem because the state ordering is fundamental to the correctness of the procedure. We, thus, produce the already explained `map` to map the position of each state to its old position. The last step concerns the building of the infinitesimal generator from the formula (4.2).

5.5 Results

We present now the results obtained with different parameterizations of the model. We show the behavior of the CPU subjected to a real workload. In the end we evaluate through the cost function, the global behavior of the model. The first two graphs show the behavior of the performance measures for different values of the parameter r. Recalling that $r \in [10^{-2}, 10^{-1}, 10^{0}, 10^{1}, 10^{2}, 10^{3}]$, the results, under a intense level of workload, are:

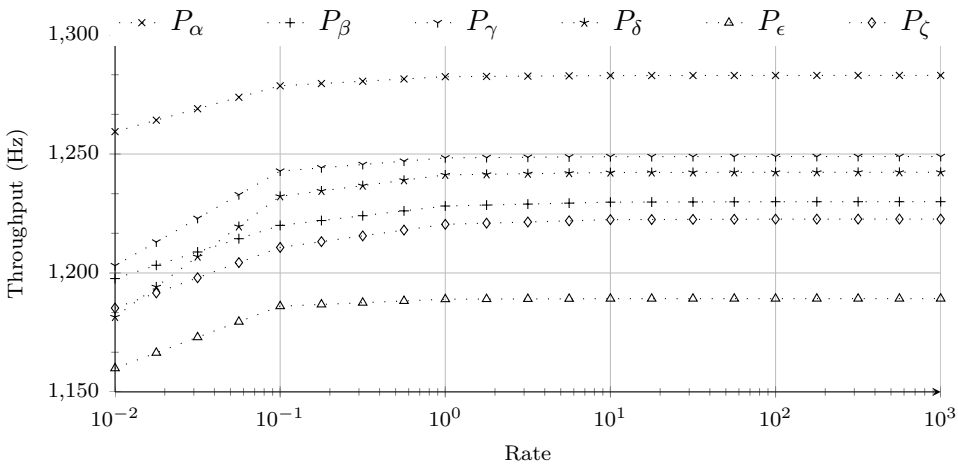

Figure 5.6: The change in the value of the throughput. It's clearly noted that, more this value increases, more throughput approaches to the maximum theoretical value (1290.35Hz, obtained via a processor model that always works at maximum speed).

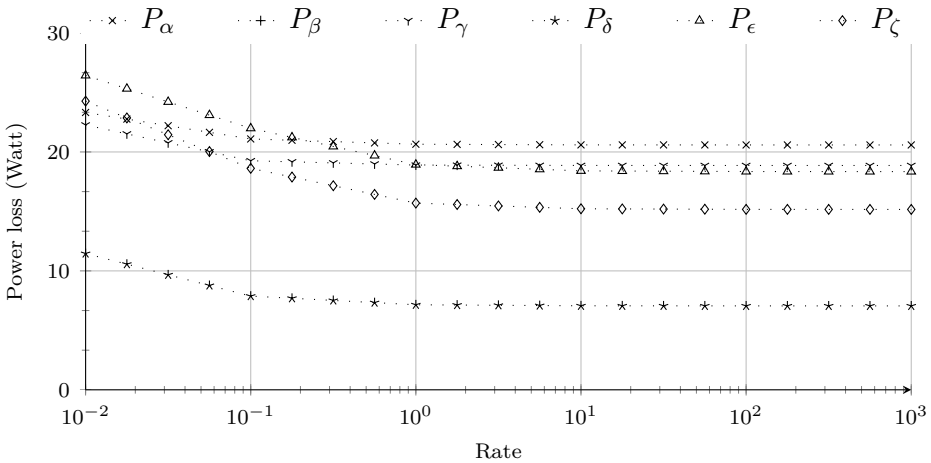

Figure 5.7: The change in the value of the power loss. It is calculated as the difference between the power actually consumed by the CPU *and that consumed in a ideal frequency level. The maximum theoretical power loss is* $63.3727 Watt$.

Throughput

We show now the bar graph that displays the variation of the level of throughput. We include only the values obtained with parameter $r = [10^{-2}, 10^{-1}, 10^{2}, 10^{3}]$. It should be noted that the value $r = 10^3$ is the best achievable result.

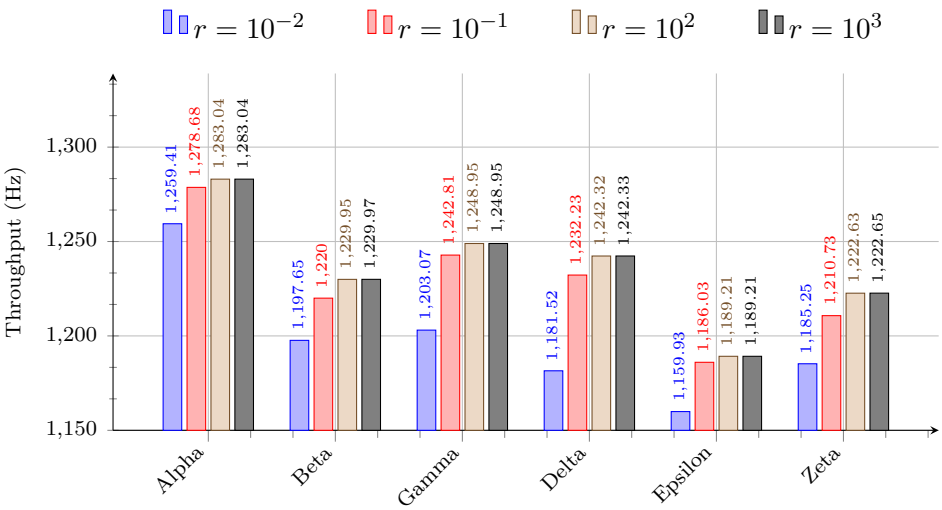

Figure 5.8: Throughput level on Google workload. The graph shows that the best performance in terms of throughput are achieved by the model P_α.

Power loss

We now show the bar graph that showing the variation of the level of power loss. We also show the percentage of improvement for this measure compared to the level of power loss maximum equal to 63.3727Watt.

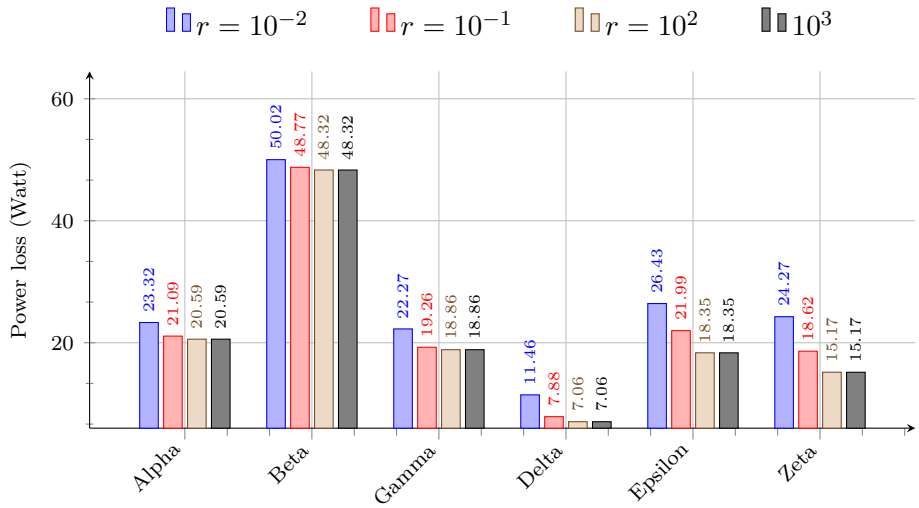

Figure 5.9: Power loss on Google workload. The graph shows that the increase of the frequency levels results in the decrease of power loss.

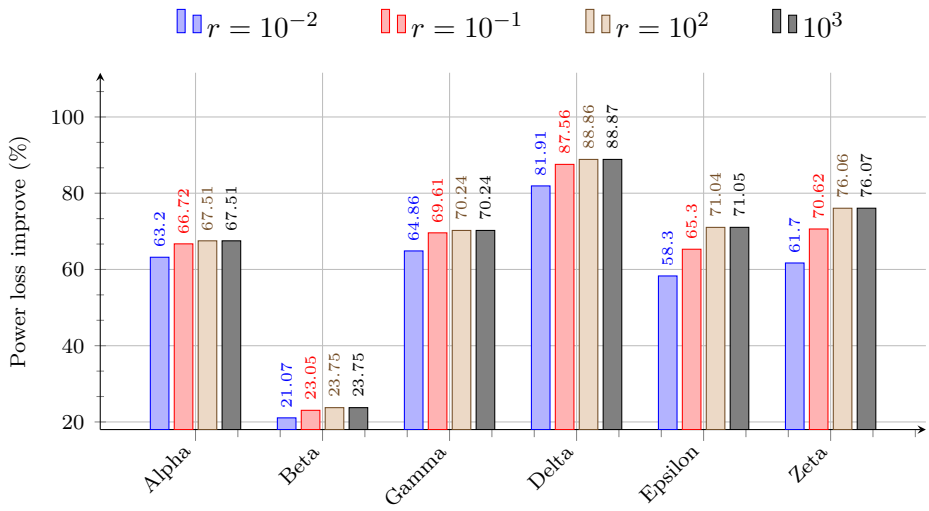

Figure 5.10: Power loss variations on Google workload. The graph shows that the best performance in terms of power loads are achieved by the model P_δ.

Cost function

We are now ready to see a comprehensive assessment of the energy performance. To do this we use the cost function $C(\alpha, \beta)$ defined in section 3.3.3. The parameters α and β are respectively set to 0.3 and 0.7

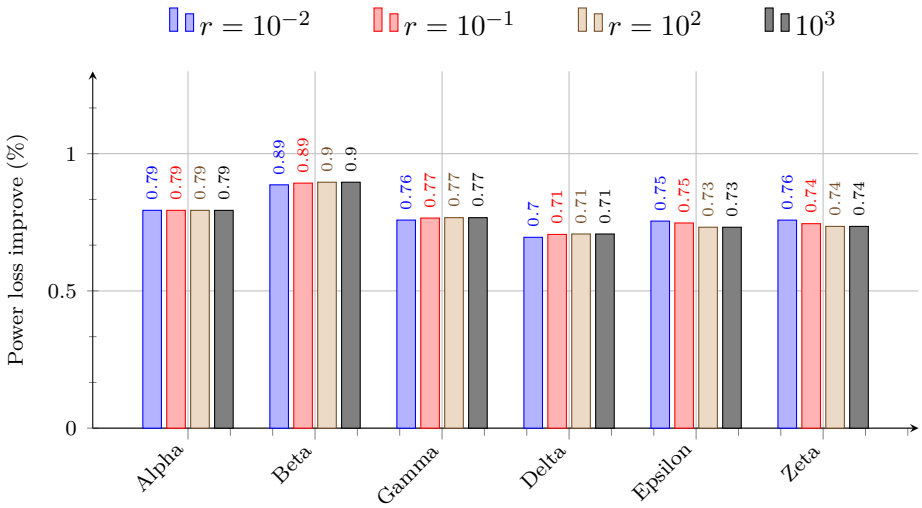

Figure 5.11: Cost function for the Google workload. The graph shows that the best frequency regulatory policy for a workload of this type is P_δ (because we are looking for the minimum value of the function).

5.6 Conclusion

In this chapter we studied the behavior of the model when it is parameterized starting from a real CPU workload trace. First, we presented the data provided by the *GoogleCluster* project giving the definition of the semantic, the format and the schema of the information it provides. Then we presented the procedure to extract the workload of each machine, explaining in the detail all its components and how it works. Finally, the model was parametrized through the infinitesimal generator resulting form the fitting of the traces in Markovian process and the results was presented.

Conclusion

In this thesis we presented a framework for the automatic evaluation of the performance of a frequency regulatory policy in term of different kind of metric.

A after a brief introduction of the formalism used, we entered in the heart of the thesis by defining the model. We formalized a workload model and a frequency model through the Performance Evaluation Process Algebra (PEPA). We also defined a set of metric used to evaluate the performance of the model such throughput and power consumption. Immediately after we test the model starting from a synthetic parametrization of the workload behavior on a set of six different regulatory policies. Finally a case study has been presented starting from the data provided by the *GoogleClusterData* project.

Extensive testing highlights how the proposed model provides important opportunities to reduce the power consumption of CPUs while maintaining reasonable level of throughput. In particular, what is appropriate to highlight is the ability to represent this kind of situations and also the relative ease in obtaining performance measures rapidly and in a purely algorithmic way. However, what we want to say is that the proposed framework is only a first step toward the modeling of complex systems. Should not be taken as a finished product but as a starting point to stimulate research toward a more conscious and intelligent use of energy resources.

There are still many things to see and prove. We, however, intend to further develop our approach in several ways. First of all we feel the need of a better formalization of the model itself. In particular by extending it increasing the number of the states both of the workload and the frequency model. In this thesis, we restrict the analysis to a four states workload model and to a maximum three states for the frequency model. This only for simplicity reason. But we must consider that a modern CPU has even ten *p-states*. Also the limitations on the workload state number is too strict. Specifically, we think that our work on the performance evaluation could be significantly improved. Same reasoning could be made on the assumption of a single-core CPU. An extension on the examination of a dual and multi core CPU should be made.

In addition we think that the fitting procedure should be improved by exploring other ways as the Hidden Markov Models [3] and the Markovian Arrival Process [21]. Finally an analysis of the behavior of real frequency regulatory policies should be done in order to verify and detect opportunities in the reduction of power consumption in a larger set of devices such as mobile devices and tablets.

Appendix A

Matlab source code

```
close all;
clear all;
format short e;
clc;

r = 6;
s = 4;
folder = 'TR001';

experiments = {'alpha' 'beta' 'gamma' 'delta' 'epsilon' 'zeta'};
exp.alpha   = [1 1 1 2; 1 1 1 2];
exp.beta    = [1 2 2 2; 1 2 2 2];
exp.gamma   = [1 1 2 2; 1 1 2 2];
exp.delta   = [1 1 2 2; 1 1 2 3; 2 2 2 3];
exp.epsilon = [1 3 3 3; 1 2 3 3; 2 2 3 3];
exp.zeta    = [1 3 3 3; 1 2 2 3; 2 2 2 3];

if (~strcmp(folder, 'Google'))
    map = [1 2 3 4]';
end

%% External libraries %%
fprintf('Automatic procedure started...\n\n');
fprintf('  1)  Activating external libraries...');
addpath(genpath(sprintf('../PEPA/Experiment/%s', folder)));
fprintf(' done\n\n');

%% Data loading %%
fprintf(sprintf('  2)  Importing data from file ../PEPA/Experiment/%s/trace.mat
    ...', folder));
load trace.mat;
fprintf(' done\n\n');

%% Fitting procedure %%
if (strcmp(folder, 'Google'))
    fprintf('  3)  Starting the automatic fitting of trace in Markovian process
        ...\n');
```

```
36      trc.original = trace(:, 1);
37      [trc.firstOrder, trc.secondOrder, Q, map, k] = model_function_fitting(trc.
            original, s, 3, 0.2);
38      [trc.fittedTime, trc.fittedValue] = model_function_simulation(Q, size(trc.
            original, 1));
39
40      for i=1:size(trc.fittedValue, 1)
41          trc.fittedValue(i, 1) = map(trc.fittedValue(i, 1), 1) - 1;
42      end
43
44      fprintf('\n');
45      clear trace;
46  else
47      fprintf('  3)  Infinitesimal generator loaded...\n');
48      [trc.fittedTime, trc.fittedValue] = model_function_simulation(Q, 4000);
49      k = 4;
50      fprintf('\n');
51  end
52
53  %% PEPA model generation %%
54  fprintf('  4)  Generating the PEPA models...\n');
55  for i=1:size(experiments, 2)
56      fprintf('      %d)  Experiment %s...', i, strcat(experiments{1, i}));
57
58      filename = sprintf('../PEPA/Experiment/%s/models/experiment.%s.pepa',
            folder, strcat(experiments{1, i}));
59      if exist(filename, 'file') ~= 0
60          delete(filename);
61      end
62
63      model_function_pepa_generator(Q, eval(sprintf('exp.%s;', strcat(experiments
            {1, i}))), map, k, filename);
64      fprintf(' done\n');
65  end
66  clear i filename;
67  fprintf('\n');
68
69  %% Waiting for Steady state distribution files
70  exit = false;
71  exMAT = zeros(size(experiments, 2), r);
72  for i=1:size(experiments, 2)
73      for j=1:r
74          fileSpace = sprintf('../PEPA/Experiment/%s/analisys/experiment.%s.R%d.
                statespace', folder, strcat(experiments{1, i}), j);
75          if exist(fileSpace, 'file') ~= 0
76              delete(fileSpace);
77          end
78
79          fileGen = sprintf('../PEPA/Experiment/%s/models/experiment.%s.R%d.
                generator', folder, strcat(experiments{1, i}), j);
80          if exist(fileGen, 'file') ~= 0
81              delete(fileGen);
82          end
```

```
    end
end

fprintf('  5)  Waiting for steady state files...');
while (~exit)
    fprintf('.');
    for i=1:size(experiments, 2)
        for j=1:r
            file = sprintf('../PEPA/Experiment/%s/analisys/experiment.%s.R%d.
                statespace', folder, strcat(experiments{1, i}), j);
            if exist(file, 'file')
                exMAT(i, j) = 1;
            end
        end
    end

    if (size(find(exMAT == 0), 1) == 0)
        exit = true;
    else
        pause(5);
    end
end
clear i j exit exMAT file fileSpace fileGen map;
fprintf(' done\n\n');

%% Cleaning of the Steady state distribution files
fprintf('  6)  Cleaning the steady state files...');
P = cell(size(experiments, 2), r);
for i=1:size(experiments, 2)
    for j=1:r
        fileSpace = sprintf('../PEPA/Experiment/%s/analisys/experiment.%s.R%d.
            statespace', folder, strcat(experiments{1, i}), j);
        P{i, j} = model_function_probability_clean(fileSpace, s);

        if exist(fileSpace, 'file') ~= 0
            delete(fileSpace);
        end
    end
end
clear i j fileSpace r exp k folder;
fprintf(' done\n\n');

%% Performance calculator
fprintf('  7)  Computing the performance indexes...');
Throughput = zeros(size(P, 1), size(P, 2));
PowerConsumed = zeros(size(P, 1), size(P, 2));
PowerLoss = zeros(size(P, 1), size(P, 2));
for i=1:size(P, 1)
    for j=1:size(P, 2)
        [Throughput(i, j), PowerConsumed(i, j), PowerLoss(i, j)] =
            model_function_performance(P{i, j}, strcat(experiments{1, i}), s);
    end
end
```

```
clear i j experiments s;
fprintf(' done\n');
```

Listing A.1: Source code of the `model_analysis` script. Main script for the analysis of performance indexes of workload models with a Markovian behavior.

```
function [ft, st, Q, map, k] = model_function_fitting(trace, states, samples,
    lambda)
% DESCRIPTION: Automatic fitting of trace into a Markov Process.

    %% Initialization %%
    fprintf('      a)  Procedure initialization...');
    ft = ones(size(trace, 1), 1);
    st = ones(size(trace, 1), 1);
    sumWindows = ones(size(trace, 1), 2);
    fprintf(' done\n');

    %% Subdivision of workload in levels %%
    fprintf('      b)  Workload subdivision...');
    s = linspace(min(trace), max(trace), states + 1)';
    fprintf(' done\n');

    %% First order trace computation %%
    fprintf('      c)  Computing the levels trace...');
    for i=1:states
        ft(trace >= s(i, 1) & trace < s(i + 1, 1)) = i - 1;
    end
    fprintf(' done\n');

    %% Sums computation %%
    fprintf('      d)  Computing the windows sums...');
    for i=1:size(ft, 1)
        if (i == 1)
            sumWindows(i, 1) = ft(i, 1);
        elseif (i <= samples)
            sumWindows(i, 1) = sum(ft(1:(i - 1), 1), 1);
        elseif (i > samples)
            sumWindows(i, 1) = sum(ft((i - samples):(i - 1), 1), 1);
        end

        sumWindows(i, 2) = ft(i, 1);
    end
    fprintf(' done\n');

    %% Second order trace computation %%
    fprintf('      e)  Computing the second order traces...');
    for i=1:size(sumWindows, 1)
        st(i, 1) = (sumWindows(i, 1) * states) + sumWindows(i, 2);
    end
    fprintf(' done\n');

    %% Probability matrix computation %%
    fprintf('      f)  Preparing the probability matrix...');
```

```
    k = ((states - 1) * (samples * states + 1)) + 1;
    K = zeros(k);

    for i=1:size(st, 1) - 1
        iState = st(i, 1) + 1;
        jState = st(i + 1, 1) + 1;

        K(iState, jState) = K(iState, jState) + 1;
    end

    n = sum(K, 2);
    for i=1:k
        for j=1:k
            if (n(i, 1) ~= 0)
                K(i, j) = K(i, j) / n(i, 1);
            end
        end
    end
    fprintf(' done\n');

    %% Map computation %%
    fprintf('      g)  Mapping computation...');
    map = zeros(size(K, 1) - size(find(n == 0), 1), 1);

    valid = 1;
    for i=1:size(K, 1)
        if (n(i, 1) ~= 0)
            map(valid, 1) = i;
            valid = valid + 1;
        end
    end
    fprintf(' done\n');

    %% Final matrix computation %%
    fprintf('      h)  Final probability matrix computation...');

    final = zeros(valid - 1);

    for i=1:(valid - 1)
        for j=1:(valid - 1)
            final(i, j) = K(map(i, 1), map(j, 1));
        end
    end
    fprintf(' done\n');

    %% Infinitesimal generator computation
    fprintf('      i)  Producing the infinitesimal generator...');
    Q = (final - eye(size(final, 1))) * lambda;
    fprintf(' done\n');
end
```

Listing A.2: Source code of the `model_function_fitting` function. It performs the fitting of the Google workload trace in Markov process.

```
function [t, y] = model_function_simulation(Q, d)
% DESCRIPTION: simulate a continuos-time Markov chain

    %% Random number generator initialization
    rng shuffle;
    s = rng;

    %% Variables initialization
    k = 1;
    t(1, k) = 0;
    y(1, k) = 1;
    time = 0;

    while time < d
        P = 1./Q(y(1, k), :);
        P(P <= 0) = inf;
        E = exprnd(P, [1 size(Q, 1)]);

        [C, I] = min(E);

        t(k + 1) = t(k) + C;
        y(k + 1) = I;

        time = t(k + 1);
        k = k + 1;
    end

    %% Random number generator state restoring
    rng(s);
end
```

Listing A.3: Source code of the `model_function_simulation` function. It performs a simulation starting from the infinitesimal generator Q of the continuos-time Markov chain.

```
function [] = model_function_pepa_generator(Q, L, map, k, filename)
% DESCRIPTION: Automatic building of PEPA files.

    %% Variables initialization
    S = zeros(1, size(Q, 2));
    w = floor(k / 4);
    fid = fopen(filename, 'w');

    %% Rates declarations
    fprintf(fid, '// Matrix rates\n');
    for i=1:size(Q, 1)
        for j=1:size(Q, 1)
            if ~(i == j || Q(i, j) == 0)
                fprintf(fid, 'q%d%d = %6.5f;\n', map(i, 1), map(j, 1), Q(i, j))
                    ;
            end
        end
    end
    fprintf(fid, 'rate = %6.5f;\n', 1);
```

```
19
20      %% CPU Workload model
21      fprintf(fid, '\n// CPU workload model\n');
22      for i=1:size(Q, 1)
23          if ~(size(find(Q(i, :) == 0)) == size(Q, 1))
24              fprintf(fid, 'WL%d = ', map(i, 1));
25
26              prn = 0;
27              for j=1:size(Q, 1)
28                  if (Q(i, j) ~= 0)
29                      if (prn > 0 && j <= size(Q, 1))
30                          fprintf(fid, ' + ');
31                      end
32
33                      if (i == j) % Sync
34                          fprintf(fid, '(w%d, rate).WL%d', map(i, 1), map(i, 1));
35                          S(1, i) = map(i, 1);
36                          prn = prn + 1;
37                      else        % Elem
38                          fprintf(fid, '(x, q%d%d).WL%d', map(i, 1), map(j, 1),
                                map(j, 1));
39                          prn = prn + 1;
40                      end
41                  end
42              end
43
44              fprintf(fid, ';\n');
45          end
46      end
47
48      %% CPU Frequency model
49      fprintf(fid, '\n// CPU frequency model\n');
50      for r=1:size(L, 1)
51          fprintf(fid, 'FR%d = ', r);
52          prn = 0;
53
54          for t=1:size(L, 2)
55              if (prn > 0 && t <= size(Q, 1))
56                  fprintf(fid, ' + ');
57              end
58
59              prt = 0;
60              for s=((t * w) - (w - 1)):(t * w)
61                  els = find(map == s);
62
63                  if (els ~= 0)
64                      if (prt > 0 && s <= (t * w))
65                          fprintf(fid, ' + ');
66                      end
67
68                      fprintf(fid, '(w%d, T).FR%d', map(els, 1), L(r, t));
69                      prt = prt + 1;
70                  end
```

```
            end

            prn = prn + 1;
        end

        fprintf(fid, ';\n');
    end

    %% System equations
    fprintf(fid, '\n// System equations\n');
    fprintf(fid, 'WL%d[%2.1f] <', map(1, 1), 1);

    for k=1:size(S, 2)
        if (k > 1 && k <= size(S, 2))
            fprintf(fid, ', ');
        end

        fprintf(fid, 'w%d', map(k, 1));
    end

    fprintf(fid, '> FR%d[%2.1f]', 1, 1);
    fclose(fid);
end
```

Listing A.4: Source code of the `model_function_pepa_generator` function. The procedure takes care of translating into process algebra the models defined by the matrixes `Q` and `L`. The `.pepa` file so built is directly interpretable by the PEPA *Eclipse plugin.*

```
function [PM] = model_function_probability_clean(filename, states)
% DESCRIPTION: clean a file and compute the steady-state probability.

    %% Open .statespace file and copy datas
    fid = fopen(filename, 'r');
    data = textscan(fid, '%s', 'delimiter', '\n', 'whitespace', '');
    strData = data{1};
    fclose(fid);

    %% Clean the .statespace file and put the data in a .csv file
    file = sprintf('%s.csv', strrep(filename, '.statespace', ''));
    fid = fopen(file, 'w');
    for i=1:size(strData, 1)
        l = strrep(strrep(strrep(strrep(regexprep(strData(i), sprintf('%d,', i)
            , '', 'once'), '{', ''), '}', ''), 'WL', ''), 'FR', '');
        fprintf(fid, '%s\n', l{1, 1});
    end
    fclose(fid);

    %% Import the .csv file, map the states and sum the steady probability
    data = importdata(file, ',');
    PM = zeros(states, max(data(:, 2)));

    for i=1:size(data, 1)
        idx = mod((data(i, 1) - 1), states) + 1;
```

```
        PM(idx, data(i, 2)) = PM(idx, data(i, 2)) + data(i, 3);
    end

    if exist(file, 'file') ~= 0
        delete(file);
    end
end
```

Listing A.5: Source code of the `model_function_probability_clean` function. The `.statespace` files produced by the PEPA *Eclipse plugin must be processed before proceeding with the derivation of the performance measures.*

```
function [TR, PC, PL] = model_function_performance(M, e, s)
% DESCRIPTION: compute the performance indexes.

    %% Initialization
    w = size(M, 1);
    f = size(M, 2);

    Prif = 100;
    Frif = 2000;

    Tr = zeros(w, f);
    Pc = zeros(w, f);
    Pl = zeros(w, f);

    WL = linspace(500, 2000, s);

    switch e
        case {'alpha'}
            FR = [1500 2000];
        case {'beta'}
            FR = [500 2000];
        case {'gamma'}
            FR = [1000 2000];
        case {'delta'}
            FR = [1000 1500 2000];
        case {'epsilon'}
            FR = [500 1000 2000];
        case {'zeta'}
            FR = [500 1500 2000];
    end

    %% Filling the matrixes
    for wl=1:w
        for fl=1:f

            % Power consumed
            Pc(wl, fl) = Prif * (FR(1, fl) / Frif)^3 * M(wl, fl);

            if (WL(1, wl) < FR(1, fl))
                % Throughput
                Tr(wl, fl) = WL(1, wl) * M(wl, fl);
```

```
                % Power loss
                %Pl(wl, fl) = Pc(wl, fl) - (Prif * (WL(1, wl) / Frif)^3 * M(wl, fl));

                Pl(wl, fl) = ((Prif * M(wl, fl)) / Frif^3) * (FR(1, fl)^3 - WL(1, wl)^3);
            else
                % Throughput
                Tr(wl, fl) = FR(1, fl) * M(wl, fl);

                % Power wasted
                Pl(wl, fl) = 0;
            end
        end
    end

    %% Performace indexes
    TR = sum(Tr(:));
    PC = sum(Pc(:));
    PL = sum(Pl(:));

end
```

Listing A.6: Source code of the `model_function_performance` function. The indexes are computed by assigning a frequency level to each state of the model and then applying the defined reward structures.

Appendix B

Workload extraction source code

```
#!/bin/bash

SRCDIR=task_usage
DSTDIR=task_usage_machine_trace

CSVDIR=task_usage_csv
LOADIR=task_usage_machine_load
PARDIR=$1

JAVASPLITTER=java_splitter
JAVASAMPLER=java_sampler

rm -f $JAVASPLITTER.class
rm -f $JAVASAMPLER.class

if [ ! -f $JAVASPLITTER.class ]; then
    javac $JAVASPLITTER.java
    echo $JAVASPLITTER.java compiled succefully!!
fi

if [ ! -f $JAVASAMPLER.class ]; then
    javac $JAVASAMPLER.java
    echo $JAVASAMPLER.java compiled succefully!!
fi

if [ ! -d "$CSVDIR" ]; then
    mkdir $CSVDIR
    echo $CVSDIR created succefully!!

    echo Purge procedure started on directory: $SRCDIR
    for filename in $(find $SRCDIR -type f -name '*.gz'); do

        GZname=${filename/$SRCDIR\//}

        CSVname=${GZname/.gz/}
        TMPname="tmp-"$CSVname
        CLNname="clean-"$CSVname
```

```
38          ORDname="ordered-"$CSVname
39
40          echo --- Extracting file $TMPname ---
41          gunzip -c $filename > $CSVDIR/$TMPname
42
43          echo --- Cleaning file $TMPname ---
44          cut -d , -f 1,2,5,6,14 $CSVDIR/$TMPname > $CSVDIR/$CLNname
45
46          echo --- Ordering file column ---
47          sed 's/,/ /g' $CSVDIR/$CLNname  | awk '{print $3,$4,$5,$1,$2}' | sed 's
                / /,/g' > $CSVDIR/$ORDname
48
49          echo --- Sorting file by machine identifier
50          sort -nt, -k1 $CSVDIR/$ORDname > $CSVDIR/$CSVname
51
52          echo --- Deleting temporary files ---
53          rm -f $CSVDIR/$TMPname $CSVDIR/$CLNname $CSVDIR/$ORDname
54      done
55  fi
56
57  if [ ! -d "$LOADIR" ]; then
58      mkdir $LOADIR
59      echo $LOADIR created succefully!!
60
61      echo Splitting procedure started on directory: $SRCDIR
62      for filename in $(find $CSVDIR -type f -name '*.csv'); do
63
64          CSVname=${filename/$CSVDIR\//}
65
66          echo --- Splitting file $CSVname ---
67          java $JAVASPLITTER $CSVDIR/$CSVname $LOADIR
68      done
69
70      rm -f $JAVASPLITTER.class
71  fi
72
73  if [ ! -d "$DSTDIR" ]; then
74      mkdir $DSTDIR
75      echo $DSTDIR created succefully!!
76  fi
77
78  echo Sampling procedure started on machine: $PARDIR
79  for filename in $(find $PARDIR -type f -name '*.csv'); do
80
81      CSVname=${filename/$SRCDIR\//}
82
83      echo --- Sampling file $CSVname ---
84      java $JAVASAMPLER $PARDIR/$CSVname $DSTDIR
85  done
86
87  rm -f $JAVAFILE.class
```

Listing B.1: Source code of the `csvfile-cleaner` bash script. Main script for the extrapolation of the workload form a specific machine of the Google cluster.

```

import java.io.*;

public class java_splitter {

    public static void main(String[] arg) throws Exception {

        FileReader rw = new FileReader(arg[0]);
        BufferedReader srcfile = new BufferedReader(rw);

        FileWriter fw = null;
        BufferedWriter dstfile = null;

        String r = srcfile.readLine();
        String cID = "";

        while (r != null) {
            String[] d = r.split(","); // d[0] -> machine identifier

            if (!cID.equals(d[0])) {
                String prefix = d[0].length() > 2 ? d[0].substring(0, 2) : d
                    [0];

                File f = new File(arg[1] + "/" + prefix + "/" + d[0] + ".csv");
                cID = d[0];

                if (dstfile != null && fw != null) {
                    dstfile.close();
                    fw.close();
                }

                fw = new FileWriter(f, true);
                dstfile = new BufferedWriter(fw);
            }

            dstfile.write(r);
            dstfile.newLine();
            r = srcfile.readLine();
        }

        srcfile.close();
        rw.close();
        System.gc();
    }
}
```

Listing B.2: Source code of the `java_splitter` java class.

```

```

```
2  import java.io.*;
3  import java.util.*;
4
5  public class java_sampler {
6
7      public static void main(String[] arg) throws Exception {
8
9          FileReader rw = new FileReader(arg[0]);
10         BufferedReader srcfile = new BufferedReader(rw);
11
12         String[] fn = arg[0].split("/");
13
14         FileWriter fw = new FileWriter(arg[1] + "/" + fn[2].substring(0, fn[2].
               length() - 4) + "_load.csv");
15         BufferedWriter dstfile = new BufferedWriter(fw);
16
17         SortedMap<Integer, Float> map = new TreeMap<Integer, Float>();
18         String r = srcfile.readLine();
19
20         Float e = null, ts = null, te = null;
21
22         while (r != null) {
23             String[] d = r.split(",");
24             e = Float.parseFloat(d[1]); // Cpu usage
25
26             ts = new Float(Long.parseLong(d[3]) / (60 * Math.pow(10, 6)));
27             te = new Float(Long.parseLong(d[4]) / (60 * Math.pow(10, 6)));
28
29             int is = ((int) Math.floor(ts / 5)) * 5;
30             int ie = ((int) Math.floor(te / 5)) * 5;
31
32             if ((ie - is) >= 0 || (ie - is) <= 5) {
33
34                 Float sum = map.get(is);
35
36                 if (sum == null) {
37                     map.put(is, e);
38                 } else {
39                     map.put(is, (e + sum));
40                 }
41             } else {
42                 System.out.println(is + "::" + ie);
43             }
44
45             r = srcfile.readLine();
46         }
47
48         Iterator<Integer> it = map.keySet().iterator();
49         while (it.hasNext()) {
50             Object key = it.next();
51
52             dstfile.write(key + "," + map.get(key));
53             dstfile.newLine();
```

```
54          }
55
56          dstfile.close();
57          fw.close();
58
59          srcfile.close();
60          rw.close();
61
62          System.gc();
63      }
64  }
```

Listing B.3: Source code of the `java_sampler` java class.

Bibliography

[1] Acpi, *Advanced configuration and Power interface specification*, 5.0 ed., December 2011.

[2] A. Acquaviva, L. Benini, and B. Riccò, *Processor frequency setting for energy minimization of streaming multimedia application*, in CODES, 2001, pp. 249–253.

[3] L. Baum and T. Petrie, *Statistical inference for probabilistic functions of finite state markov chains*, The Annals of Mathematical Statistics, 37 (1966), pp. 1554–1563.

[4] J. Bergstra and J. W. Klop, *Algebra of communicating processes with abstraction*, Theoretical Computer Science, 37 (1985).

[5] G. Bolch, S. Greiner, H. De Meer, and K. Trivedi, *Queueing Networks and Markov Chains*, Wiley-Blackwell, 2nd edition ed., 2006.

[6] A. Carroll and G. Heiser, *An analysis of power consumption in a smartphone*, in Proceedings of the 2010 USENIX conference on USENIX annual technical conference, 2010.

[7] M. Chains, F. Exponential, and S. F. Automata, *Note : Maximum likelihood estimation for markov chains derivation of the mle for markov chains*, Spring, 6 (2009), pp. 1–8.

[8] K. Choi, K. Dantu, W.-C. Cheng, and M. Pedram, *Frame-based dynamic voltage and frequency scaling for a mpeg decoder*, in ICCAD, 2002, pp. 732–737.

[9] B. Dietrich and S. Chakraborty, *Managing power for closed-source android os games by lightweight graphics instrumentation*, in NetGames, 2012.

[10] G. Fettweis and E. Zimmermann, *Ict energy consumption-trends and challenges*, in Proceedings of the 11th International Symposium on Wireless Personal Multimedia Communications, vol. 2, 2008, p. 6.

[11] L. Gallina, S. Hamadou, A. Marin, and S. Rossi, *A framework for throughput and energy efficiency in mobile ad hoc networks*, in Wireless Days, 2011, pp. 1–6.

[12] S. Gilmore and J. Hillston, *The pepa workbench: A tool to support a process algebra-based approach to performance modelling*, in Computer Performance Evaluation, 1994.

[13] J. Hillston and L. Kloul, *An efficient kronecker representation for pepa models*, in PAPM-PROBMIV, 2001.

[14] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Inc., 1985.

[15] R. Howard, *Dynamic Probabilistic Systems*, vol. 2, John Wiley and Sons, 1971.

[16] S. Karlin and H. Taylor, *An introduction to Stochastic modeling*, Academic Press, 3 ed., 1998.

[17] L. Kleinrock, *Queueing Systems*, vol. 1, Wiley-Interscience, 1975.

[18] G. Lawler, *Introduction to Stochastic Processes*, Chapman and Hall/CRC, 2 ed., 2006.

[19] R. Milner, *Communication and concurrency*, Prentice Hall, 1989.

[20] S. Murugesan, *Harnessing green it: Principles and practices*, IT professional, 10 (2008), pp. 24–33.

[21] M. Neuts, *Models based on the markovian arrival process*, IEICE Transactions on Communications, 75 (1992), pp. 1255–1265.

[22] B. Plateau and K. Atif, *Stochastic automata network for modeling parallel systems*, IEEE Trans. Software Eng., 17 (1991).

[23] T. Rauber and G. Rünger, *Energy-aware execution of fork-join-based task parallelism*, in MASCOTS, 2012.

[24] C. Reiss, J. Wilkes, and J. L. Hellerstein, *Google cluster-usage traces: format + schema*, technical report, Google Inc., Mountain View, CA, USA, Nov. 2011. Revised 2012.03.20. Posted at URL http://code.google.com/p/googleclusterdata/wiki/TraceVersion2.

[25] W. J. Stewart, *Probability, Markov Chains, Queues and Simulation*, Princeton University Press, 2009.

[26] M. Tribastone, A. Duguid, and S. Gilmore, *The pepa eclipse plugin*, SIGMETRICS Performance Evaluation Review, 36 (2009).

[27] M. Weiser, B. B. Welch, A. J. Demers, and S. Shenker, *Scheduling for reduced cpu energy*, in OSDI, 1994, pp. 13–23.

[28] W. Yuan and K. Nahrstedt, *Practical voltage scaling for mobile multimedia devices*, in ACM Multimedia, 2004, pp. 924–931.

[29] J. Zhuo and C. Chakrabarti, *Energy-efficient dynamic task scheduling algorithms for dvs systems*, ACM Trans. Embedded Comput. Syst., 7 (2008).