Dynamic Power Management for QoS-Aware Applications

Moreno Marzolla **Raffaela Mirandola**

Technical Report UBLCS-2011-04

June 2011

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)

The University of Bologna Department of Computer Science Research Technical Reports are available in PDF and gzipped PostScript formats via anonymous FTP from the area `ftp.cs.unibo.it:/pub/TR/UBLCS` or via WWW at URL `http://www.cs.unibo.it/`. Plain-text abstracts organized by year are available in the directory `ABSTRACTS`.

Recent Titles from the UBLCS Technical Report Series

2009-10 *Probabilistic Data Integration*, Magnani, M., Montesi, D., March 2009.

2009-11 *Equilibrium Selection via Strategy Restriction in Multi-Stage Congestion Games for Real-time Streaming*, Rossi, G., Ferretti, S., D'Angelo, G., April 2009.

2009-12 *Natural deduction environment for Matita*, C. Sacerdoti Coen, E. Tassi, June 2009.

2009-13 *Hints in Unification*, Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E., June 2009.

2009-14 *A New Type for Tactics*, Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E., June 2009.

2009-15 *The k-Lattice: Decidability Boundaries for Qualitative Analysis in Biological Languages*, Delzanno, G., Di Giusto, C., Gabbrielli, M., Laneve, C., Zavattaro, G., June 2009.

2009-16 *Landau's "Grundlagen der Analysis" from Automath to lambda-delta*, Guidi, F., September 2009.

2010-01 *Fast overlapping of protein contact maps by alignment of eigenvectors*, Di Lena, P., Fariselli, P., Margara, L., Vassura, M., Casadio, R., January 2010.

2010-02 *Optimized Training of Support Vector Machines on the Cell Processor*, Marzolla, M., February 2010.

2010-03 *Modeling Self-Organizing, Faulty Peer-to-Peer Systems as Complex Networks* Ferretti, S., February 2010.

2010-04 *The qnetworks Toolbox: A Software Package for Queueing Networks Analysis*, Marzolla, M., February 2010.

2010-05 *QoS Analysis for Web Service Applications: a Survey of Performance-oriented Approaches from an Architectural Viewpoint*, Marzolla, M., Mirandola, R., February 2010.

2010-06 *The dark side of the board: advances in Kriegspiel Chess (Ph.D. Thesis)*, Favini, G.P., March 2010.

2010-07 *Higher-Order Concurrency: Expressiveness and Decidability Results (Ph.D. Thesis)*, Perez Parra, J.A., March 2010.

2010-08 *Machine learning methods for prediction of disulphide bonding states of cysteine residues in proteins (Ph.D. Thesis)*, Shukla, P., March 2010.

2010-09 *Pseudo-Boolean clustering*, Rossi, G., May 2010.

2010-10 *Expressiveness in biologically inspired languages (Ph.D. Thesis)*, Vitale, A., March 2010.

2010-11 *Performance-Aware Reconfiguration of Software Systems*, Marzolla, M., Mirandola, R., May 2010.

2010-12 *Dynamic Scalability for Next Generation Gaming Infrastructures*, Marzolla, M., Ferretti, S., D'Angelo, G., December 2010.

2011-01 *Server Consolidation in Clouds through Gossiping*, Marzolla, M., Babaoglu, O., Panzieri, F., January 2011 (Revised May 2011).

2011-02 *Adaptive Approaches for Data Dissemination in Unstructured Networks*, D'Angelo, G., Ferretti, S., Marzolla, M., January 2011.

2011-03 *Distributed Computing in the 21st Century: Some Aspects of Cloud Computing*, Panzieri, F., Babaoglu, O., Ghini, V., Ferretti, S., Marzolla, M., May 2011.

Dynamic Power Management for QoS-Aware Applications

Moreno Marzolla[1] **Raffaela Mirandola**[2]

Technical Report UBLCS-2011-04

June 2011

Abstract

Reducing the power requirement of large IT infrastructures is becoming a major concern. Energy savings can be achieved with hardware and/or software solutions; in particular, modern CPUs can operate at different power levels that can be selected by software: low power modes reduce energy consumption at the cost of lowering also the CPU processing rate. In this paper we address the problem of reducing energy consumption of a large-scale distributed application which must comply to Service Level Agreements. Specifically, we propose EASY (Energy Aware reconfiguration of software SYstems), an on-line algorithm for dynamically adjusting the processing speed of individual devices in order to minimize the energy requirement and at the same time allow the application to provide an average response time below a predefined threshold. Our algorithm uses a Queueing Networks performance model to drive the reconfiguration process, so that the number of individual reconfiguration actions is greatly reduced. We formulate the energy conservation problem as a Mixed Integer Programming problem, for which we propose a heuristic solution technique. Numerical experiments show that the heuristic produces almost optimal results at a substantially lower computational cost. Therefore, Energy Aware reconfiguration of software SYstems (EASY) can be effectively applied on-line to make a large system energy-proportional.

1. Università di Bologna, Dipartimento di Scienze dell'Informazione, Mura A. Zamboni 7, I-40127 Bologna (Italy); Email: marzolla@cs.unibo.it
2. Politecnico di Milano, Dipartimento di Elettronica e Informazione, Piazza Leonardo da Vinci, I-20133 Milano, Italy; Email: mirandola@elet.polimi.it

1 Introduction

The constant growth of energy usage in industrialized countries is creating problems to the sustainability of the Earth development. According to a report by Koomey [1], the total electrical power consumed by servers worldwide doubled from 2000 to 2005, representing an annual growth rate of 14% yearly for the U.S. and 16% worldwide. The total power consumption of all servers in the U.S. in 2005 was about $2.6 \times 10^{12} W$ (2.6 billion kW); all servers in the world are estimated to consume about $7 \times 10^{12} W$. If we include also the energy required for cooling and for powering ancillary equipment, the total power consumption raises to $5.2 \times 10^{12} W$ for the U.S., and $14 \times 10^{12} W$ worldwide [1, Table 6].

Besides, as shown in [2, 3], the interest towards efficient use of technology is motivated by some alarming trends showing, for example, that computing equipment in the U.S. alone is estimated to consume more than 20 million giga-joules of energy per year, the equivalent of four-million tons of carbon-dioxide emissions into the atmosphere [3]. IT analysis firm IDC[1] estimates the total worldwide spending on power management for enterprises was likely 40 billion dollars in 2009.

The power consumption of the IT sector is becoming a serious concern in the design and operation of the hosting centers. For this reason, some new disciplines such as *green computing* are growing up to study how to build and manage computing infrastructures that require less energy still providing the same quality of service [3]. Rudimentary techniques for power management, e.g., shutting down servers, can impact the ability of the hosting center to meet the Service Level Agreements (SLAs) implicitly or explicitly stipulated with clients. Shutting down temporarily unused servers ensures the maximum energy reduction, at the cost of making those servers unable to process any request. Bringing up the machines when needed requires a significant start-up time (up to several minutes, depending also on the time needed to start the applications). Start-up delays can severely impact the ability of the system to promptly handle workload fluctuations. Besides, repeated on-off cycles can also increase the wear-and-tear of hardware components, adding costs for their procurement and re-placement [4].

It is important to consider that the average utilization of a server in a datacenter is quite low, but rarely zero: Barroso and Hölzle observe that most of the time, servers operate between 10 and 50 percent of their maximum utilization level [5]; the utilization rarely drops to zero. This behavior is not accidental, but derives from the application of appropriate engineering and provisioning principles: if the number of servers is reduced to increase their average utilization, then the system would operate near its saturation point. In Fig. 1 we show the well known relation between utilization and response time of a generic device: as the utilization approaches one, the response time diverges since an infinite queue of requests builds up; what happens in practice is that devices start dropping requests at high utilization levels, as internal buffers fill up. A certain amount of slack capacity is necessary to avoid operating the system near the saturation point, to cope with workload fluctuations.

We also observe that an important family of large scale Web applications includes the so called Online Data Intensive (OLDI) services [6] such as search engines, online advertising, Massive Multiplayer Online Games (MMOGs) and similar applications that process external requests under very strict response time constraints. OLDI workloads are driven by user-generated queries that interact with massive data sets; responses must be produced within a very short time, sometimes in the sub-second scale. Furthermore, OLDI workloads fluctuate considerably over time, usually with a periodic pattern resulting from human activity periods.

As an example, we show in Fig. 2 the number of online players connected to the RuneScape MMOG, collected by monitoring the game Web page during several weeks starting from April 2011 (more details will be given in Section 8). A daily pattern is clearly visible, which results from the overlapping of activity periods of a large population of players distributed over different time zones.

The discussion above suggests to reduce the power consumption by making the system *power proportional* [5], meaning that the power consumption of devices is kept proportional to

1. `http://www.idc.com/`

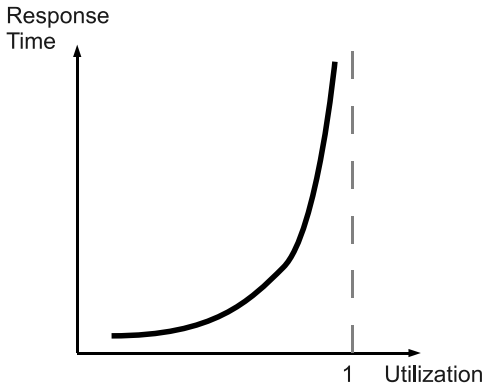


Figure 1. Response time of a device as a function of its utilization

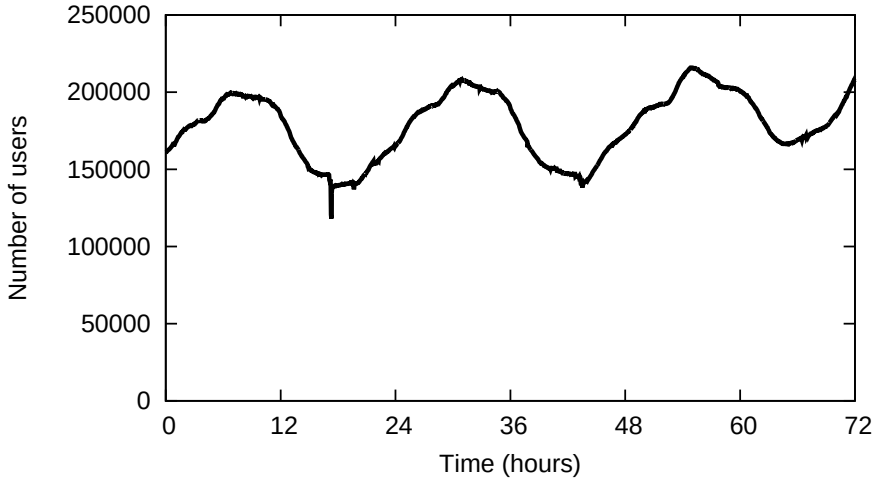


Figure 2. Number of RuneScape players

their utilization. Ideally, a device should consume zero power when its utilization is zero, and full power when its utilization is one. Unfortunately, such "perfect" power proportionality has not been achieved yet, current server processors can consume about 30% of their peak power during very low activity periods, creating a dynamic range of more than 70% their peak power. CPUs targeted at mobile devices have idle power usually reaching one tenth of their peak power.

Specifically, most processors allow dynamic frequency/voltage scaling (DVS), where the frequency (and voltage) can be lowered to produce much more savings in power consumption compared to how much one looses on quality. The Advanced Configuration and Power Interface (ACPI) specification has been proposed as an open standard for configuration and power management of individual devices or entire systems [7]. ACPI defines a larger set of low power states, some of which are fully operational, meaning that the device still processes user requests at a possibly slower rate. For processors, power savings within operational states is achieved through *dynamic frequency scaling*, in which the CPU frequency and voltage are dynamically adjusted to reduce the energy requirement. Since ACPI state changes can be controlled by software, there is a great potential for achieving significant energy reductions by implementing more elaborate power/performance tradeoffs, which were not possible with the previous generation of devices.

Contribution of this paper In this paper we pursue this direction by proposing a dynamic power management strategy which minimizes the overall power consumption of a large scale IT infrastructure subject to Quality of Service (QoS) constraints. While using dynamic voltage scaling may not provide as much savings as shutting down a server completely, the advantage is that it can still service requests (albeit at a lower frequency), and it can be switched back to full speed with negligible overhead. This is essential for OLDI applications mentioned above.

We propose EASY, a framework for dynamic power optimization of QoS-aware large-scale systems. We consider a distributed system subject hosting an OLDI application subject to response time constraints: specifically, the system response time must be kept below a given threshold. The system is made of a set of ACPI-capable devices (CPUs, disks, system modules and so on). We assume that the system is subject to variable workload; the internal load of the system can be unevenly spread across the servers, and can also shift from device to device at any time. EASY is an on-line selection of power/performance levels of devices such that the average response time is kept below the threshold, and the total energy consumption rate is minimized. EASY can be applied to any device supporting the ACPI specification (not just CPUs), hence our approach is quite general and can be applied to many different kind of systems.

EASY is based on a Monitor-Analyze-Plan-Execute control loop: the system is enhanced with a monitor component, which is responsible for collecting performance measures at runtime. This information is used both to identify when the response time constraint is being violated, and also to drive the reconfiguration process itself. The analyzer and the planner use a simple Queueing Network (QN) model to quickly analyze different power/performance settings. All quantitative parameters needed to evaluate the QN model are estimated by the monitor: hence, EASY does not require any prior knowledge on the system. Additionally, EASY can adapt to both changes in the workload, or changes in the load distribution within the devices. The allowed maximum system response time can be changed at run time if necessary, allowing system administrators to easily redefine the SLA. We formulate the energy minimization problem as a Mixed Integer Programming (MIP) problem; we describe an effective heuristic which can provide a feasible solution faster than handling the optimization problem with a conventional general-purpose solver. Using a set of simulation experiments, we show that the heuristic used by EASY is very fast, and the quality of the results is statistically comparable to the optimal solution of the MIP problem computed through a MIP solver.

Paper Organization The remainder of the paper is organized as follows. Section 2 shortly describes the ACPI specification to which we refer. Section 3 formally describes the problem which we are addressing. The high level architecture of EASY is described in Section 4; Section 5 describes the QN performance model, and how response times can be estimated from the quantitative parameters collected by the monitor. In Section 6 we formulate the optimization problem as a MIP problem, which can be handled by standard MIP solvers. In Section 7 we describe an effective heuristic which can be used to solve the same problem much more efficiently. The results of the extensive simulation experiments used to validate our approach are then presented and discussed in Section 8. Related works are summarized in Section 9, and contrasted with our approach. Finally, Section 10 presents concluding remarks, and proposes future research directions along which the approach described in this paper can be extended. To make this paper self-contained, we include in the Appendix: a table summarizing the main notation used in the paper, a short overview of Queueing Network modeling, and a description of the MIP problem using the GNU MathProg language.

2 The ACPI Specification

Most modern CPUs provide a useful feature, namely the support for the Advanced Configuration and Power Interface (ACPI). ACPI [7] is an open standard for platform-independent configuration and power management of both individual devices and entire systems. ACPI-compliant devices allow higher-level components (such as the Operating System) to explicitly control the

system power consumption. ACPI is an improvement over earlier power-saving solutions (e.g., the Advanced Power Management application programming interfaces), as it allows complex power management policies to be managed by the OS rather than a ROM BIOS. The OS can gather information at the system (hardware) and application levels in order to take better power management decisions; this would not be possible if power management features were embedded into device firmware or ROMs.

The ACPI specification defines global system states, individual device power states and processor power states. The *global states* G0–G3 are defined as follows (in order of decreasing power consumption):

G0–Working. In this state the system operates normally and application threads are executed; individual components may have their power state changed dynamically, e.g. by the user.

G1–Sleeping. In this state the system requires less power, but application threads are not being executed. The system can be brought back to working state without the need to reboot.

G2–Soft off. In this state the system power consumption is minimal. User or system software is not executed. The system must be restarted, and requires a large latency in order to go back to the working state.

G3–Mechanical Off. In this state the system is shut off through mechanical means. Except for the real-time clock, the power consumption is zero.

The *device states* D0–D3 describe the power state of individual devices attached to the system (e.g., disks drives, monitors and so on). The ACPI specification defines the device states as follows, in order of decreasing power consumption:

D0 (Fully-On). The device is active and responsive, requiring the highest power consumption.

D1, D2, D3hot. These states require less power (D0>D1>D2>D3hot), but the device is expected to preserve increasingly less context.

D3 (Off). In this state, power has been removed from the device. The device context is lost when this state is entered.

Processor power states C0–C3 are defined within the global working state G0 as follows, again in order of decreasing power consumption:

C0. The processor executes instructions normally.

C1–Halt. The processor is not executing instructions, but can return to the C0 state with minimal latency. All software-visible states are preserved.

C2–Stop-Clock. This state offers improved power savings over the C1 state, but requires a larger latency to return to an executing state. All software-visible states are preserved.

C3–Sleep. This state offers improved power savings over the C1 and C2 states. Cache coherency is not guaranteed, and must be ensured by software on resume.

Finally, the ACPI specification defines *device and processor performance states* Pi within the active states (C0 for processors and D0 for devices). Devices and processors may support up to 16 states ($P0$ to $P16$), where state Pi requires more energy than state $P(i+1)$, but provides better performance. Transition between ACPI performance states incurs negligible overhead; this is the reason why we focus on them in this paper.

Figure 3 summarizes the valid transitions between ACPI states.

Power reduction is achieved by dynamically adjusting the processor core voltage, clock rate or both, since the power P dissipated by a CMOS circuit satisfies the following relation [9]:

$$P \propto CV^2 f$$

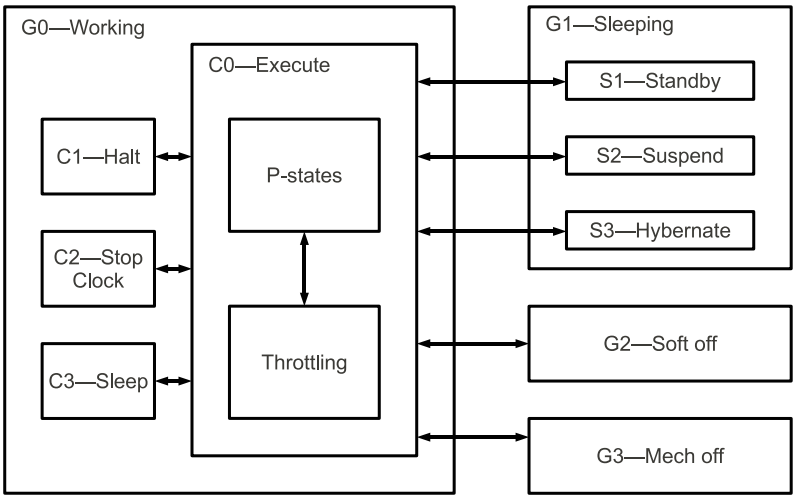


Figure 3. ACPI states [8]

P-State	Frequency	Voltage	Power
P0	1.6 GHz	1.484 V	24.5 W
P1	1.4 GHz	1.420 V	
P2	1.2 GHz	1.276 V	
P3	1.0 GHz	1.164 V	
P4	800 MHz	1.036 V	
P5	600 MHz	0.956 V	6 W

Table 1. P-States for the Intel Pentium M processor at 1.6 GHz [10, Table 1.6]; power consumption for states P1–P4 is not available in the reference

where C is the capacitance, V the voltage and f the clock frequency. Various technologies are used by different vendors to achieve dynamic power reduction, such as Intel Speedstep [10], AMD PowerNow! [11], and IBM EnergyScale [12]. Additionally, AMD CoolCore [13] is capable of turning off individual parts of the processor to further reduce energy consumption and CPU temperature.

To provide some actual figures, Table 1 reports the operating voltages, frequences and power consumption of the Pentium M processor, which was used in mobile computers due to its low power requirements (note: intermediate power consumptions are not available on the reference cited). Table 2 shows the same information for the AMD Opteron processors, which is part of the previous generation of AMD processors used in servers and desktops. Finally, Table 3 refers to the 2 GHz VIA C7-M processor, currently offered for the mobile market.

3 Problem Formulation

In this section we introduce the notation used in the rest of this paper, and we formally describe the problem addressed; the notation is also summarized in A.

We consider an application running on a distributed system made of a set $\mathcal{K} = \{1, \ldots, K\}$ of K devices; each device is uniquely identified by an integer k in the set $\mathcal{K}$. Device k can be a CPU, a disk or any kind of device. Clients interact with the application by issuing requests which are processed by the system. Processing involves requesting service from multiple devices; at the end, a response is sent back to the client.

P-State	Frequency	Voltage	Power
P0	2.8 GHz	1.40 V	92.6 W
P1	2.6 GHz	1.35 V	90.2 W
P2	2.4 GHz	1.30 V	77.0 W
P3	2.2 GHz	1.25 V	65.7 W
P4	2.0 GHz	1.20 V	55.9 W
P5	1.8 GHz	1.15 V	47.6 W
Pmin	1.0 GHz	1.10 V	36.1 W

Table 2. P-States for the AMD Opteron Processor [14, Table 11, p. 14]

P-State	Frequency	Voltage	Power
P0	2.0 GHz	1.148 V	20 W
P1	1.8 GHz	1.132 V	18 W
P2	1.6 GHz	1.100 V	15 W
P3	1.4 GHz	1.052 V	13 W
P4	1.0 GHz	1.004 V	10 W
P5	800 MHz	0.844 V	7 W
P6	600 MHz	0.844 V	6 W
P7	400 MHz	0.844 V	5 W

Table 3. P-State Information for the 2 GHz VIA C7-M Processor (Source: `http://www.via.com.tw/en/initiatives/greencomputing/powersaver.jsp`)

Device k supports $L[k]$ different performance states[2], which for notational convenience will be labelled as integers in the set $\mathcal{L}[k] = \{1, \ldots, L[k]\}$. State 1 is the state with maximum speed and higher power consumption rate, while state $L[k]$ offers the lower power consumption rate but also provides the lower performance.

We denote with $EN[k, j]$ the energy consumption rate of device $k \in \mathcal{K}$ in state $j \in \mathcal{L}[k]$; energy consumption rates can be different from device to device (we allow heterogeneous systems). The total energy consumed by device k in state j over a period of T time units is $T \times EN[k, j]$. We require that the energy consumption rates and relative speeds are monotonically decreasing:

$$EN[k, 1] > EN[k, 2] > \ldots > EN[k, L[k]] > 0$$

We denote with $RSP[k, j]$ the *relative speed* of device k in state j, defined as:

$$RSP[k, j] = \frac{\text{Processing rate of device } k \text{ in state } j}{\text{Processing rate of device } k \text{ in state } 1} \tag{1}$$

Intuitively, $RSP[k, j]$ represents the slowdown of device k running in state j with respect to the same device operating in state 1. For example, if $RSP[k, 3] = 0.2$, then a computation requiring x time units to complete on device k in state 1 would take $(1/0.2)x = 5x$ time units on the same device in state 3. Relative speeds can be different from device to device; we only require that processing rates are monotonically decreasing:

$$1 = RSP[k, 1] > RSP[k, 2] > \ldots > RSP[k, L[k]] > 0$$

This means that a device operating in higher-numbered performance states is slower than the same device in lower-numbered performance state.

2. In this paper we make extensive use of array subscripts. In order to enhance readability of subscript indexes, we write $L[k]$ instead of L_k to denote the k-th element of array L.

A *system state* (or *system configuration*) is a vector $\mathbf{S} = (S[1], \ldots, S[K])$, which represents the fact that device k is in performance state $S[k] \in \mathcal{L}[k]$.

The goal of EASY is to dynamically adjust the state of each device so that the system power consumption rate is minimized, while maintaining the (estimated) system response time below a predefined threshold $R_{\max}$. Let $\mathbf{S}(t)$ be the system configuration at time t. We want to identify a state $\mathbf{S}(t)$ which solves the following optimization problem:

$$\begin{aligned} &\mathscr{P}(R_{\max}) \equiv \qquad (2)\\ &\textbf{Minimize:} \quad E(\mathbf{S}(t)) = \sum_{k=1}^{K} EN[k, S[k](t)] \\ &\textbf{Subject to:} \quad R(\mathbf{S}(t)) \leq R_{\max} \\ &\qquad\qquad S[k](t) \in \mathcal{L}[k]\,, \quad \text{for all } k \in \mathcal{K} \end{aligned}$$

where $R(\mathbf{S}(t))$ is the system response time with configuration $\mathbf{S}(t)$. The value of the objective function $E(\mathbf{S}(t)) = \sum_{k=1}^{K} EN[k, S[k](t)]$ is the total energy consumption rate with configuration $\mathbf{S}(t)$.

It is important to observe that the response time $R(\mathbf{S}(t))$ depends on the system configuration $\mathbf{S}(t)$ at time t, but also on other parameters, including the workload intensity (number of concurrent users interacting with the system), which can fluctuate over time. Therefore, the optimization problem (2) should be solved continuously at run-time in order to allow the system to operate within the requested QoS constraints with minimum energy consumption. Of course, computing new configurations with high frequency is infeasible, as that operation would become a bottleneck. We describe in the next section a practical approach for identifying reconfiguration times and computing new system configurations.

We remark that all devices have a finite processing capacity, i.e., can process requests up to a finite maximum rate; obviously, the maximum processing capacity can be obtained when all devices operate in state 1. Nevertheless, the requests arrival rate can be larger than the maximum processing capacity, resulting in arbitrarily large delays due to queueing of requests at the bottleneck device(s). Thus, if the arrival rate of requests keeps increasing, the system response time gets larger no matter what the system configuration is. From this, we conclude that the optimization problem (2) may have no solution, e.g., when the workload intensity is higher than the maximum system capacity. In such case, EASY will return $\mathbf{S} = (1, \ldots, 1)$ as a pseudo-solution, in which all devices operate in the faster ACPI state.

4 EASY Architecture

Problem (2) represents a family of problems which are parametrized by the continuous time value t. In order to make the problem tractable, we consider discrete values of t, namely $t = 0, 1, \ldots$. Each t identifies a time interval of duration Δt. During each interval, the system configuration does not change; at the end of each period, a new configuration is identified and applied, if necessary. The duration Δt of each observation period can be tuned in order to select a trade-off between responsiveness and overhead of EASY.

Specifically, we propose a reactive system based on the Monitor-Analyze-Plan-Execute control loop shown in Fig. 4. During the *Monitor* step, EASY measures the response, throughput and individual device utilizations by collecting samples from the running system over the current time interval t. At the end of the observation period, the information collected by the monitor is used in the *Analyze* step to instantiate a performance model based on QNs. The performance model is used during the *Plan* step to quickly analyze different configurations, until an approximate solution of the optimization problem (2) is found. The solution of (2) is the new system configuration $\mathbf{S}(t+1)$ which will be applied to the next period $t+1$ by the *Execute* step.

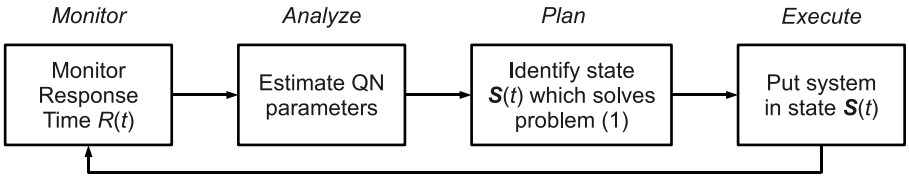


Figure 4. Monitor-Analyze-Plan-Execute control loop

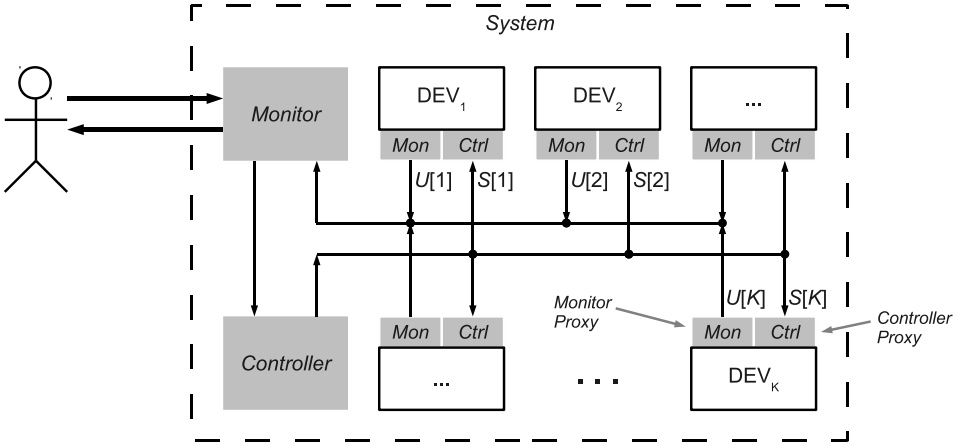


Figure 5. System architecture

To support the implementation of the control loop above, the system is enhanced with additional components, shown in gray in Fig. 5. The *monitor* is responsible for collecting performance statistics on the running system: it measures the system response time $\overline{R}(t)$ during period t, the system throughput $\overline{X}(t)$ and individual device utilizations $\overline{U}[k](t)$, under the current configuration $\mathbf{S}(t)$. System response time and throughput are measured by passive analysis of user requests; device utilizations are collected by software probes running on the devices themselves, or on proxies that can interact with the devices. The *controller* is activated at the end of observation intervals, and uses the parameters collected by the monitor to identify a new state $\mathbf{S}(t+1)$ which solves the optimization problem (2). The new states are sent to the devices as commands which trigger an ACPI state change.

It is important to remember that the monitor and the controller operate at run time, so they must be efficient in order not to become a bottleneck. In particular, the controller needs a fast way to analyze the impact of different configurations on the system response time; this rules out approaches based on "trial and error", in which some state change is attempted directly on the system. Instead, we use a product-form QN performance model [15], which possess efficient solution techniques, yet being expressive enough to accurately model many kinds of systems. We propose a simple greedy heuristic for identifying an approximate solution to the optimization problem (2) by evaluating a limited number of alternatives.

In order to avoid single point of failures, the monitor and controllers should be implemented using appropriate fault-tolerant techniques. A thoughtful discussion on implementation details is outside the scope of this paper.

The monitoring component of Fig. 5 is responsible for collecting observations on the running system. System throughput, response time, and individual device utilizations are estimated by collecting multiple samples over the observation period t, of duration Δt. If C is the number of user requests that the system processed within period t, and $t[1], t[2], \ldots, t[C]$ are the times

Algorithm 1 The EASY Algorithm

Require: Thresholds $R_{\text{low}} < R_{\text{high}} < R_{\max}$
1: Let $\mathbf{S}$ be the initial system configuration
2: **loop**
3: $t :=$ current time
4: Measure $\overline{\mathbf{U}}(t), \overline{X}(t), \overline{R}(t)$ over an interval of duration Δt
5: **if** $(\overline{R}(t) > R_{\text{high}})$ **then**
6: $\mathbf{S}' :=$ Speedup$(\mathbf{S}, \overline{\mathbf{U}}(t), \overline{X}(t), \overline{R}(t))$
7: **else if** $\left(\overline{R}(t) < R_{\text{low}}\right)$ **then**
8: $\mathbf{S}' :=$ Slowdown$(\mathbf{S}, \overline{\mathbf{U}}(t), \overline{X}(t), \overline{R}(t))$
9: Apply system state $\mathbf{S}'$
10: $\mathbf{S} := \mathbf{S}'$

taken to process the C requests, then the system response time $\overline{R}(t)$ is estimated as $\sum_{c=1}^{C} t[c]/C$, and the system throughput $\overline{X}(t)$ as $C/\Delta t$. The device k utilization $\overline{U}[k](t)$ is the fraction of the interval Δt in which device k was busy (i.e., processing some request); note that by definition, $\overline{U}[k](t) \in [0, 1]$.

At the end of the observation period, the parameters collected by the monitor are used to decide whether a new configuration should be computed. The basic idea is the following: if the measured system response time $\overline{R}(t)$ is below the threshold $R_{\max}$, we try to slow down some devices by setting them to higher numbered performance states; on the other hand, if the system response time is above $R_{\max}$, we try to speed up bottleneck devices by setting them to lower numbered performance states.

Unfortunately, if $\overline{R}(t)$ jumps above and below the threshold $R_{\max}$, the simple approach above is prone to triggering too many reconfigurations. We address this issue by using an *hysteresis* mechanism, implemented through two additional thresholds $R_{\text{low}}$and R_{high}, such that $R_{\text{low}} < R_{\text{high}} < R_{\max}$; a system reconfiguration is triggered only when $\overline{R}(t) > R_{\text{high}}$ or $\overline{R}(t) < R_{\text{low}}$.

Algorithm 1 shows the main control loop of EASY, where the hysteresis mechanism just described is actually used. At each iteration, the system response time $\overline{R}(t)$, throughput $\overline{X}(t)$ and device utilizations $\overline{U}[k](t)$ are measured after collecting samples during a time interval of duration Δt (line 4). If the response time $\overline{R}(t)$ is larger than R_{high}, EASY identifies a new configuration by switching bottleneck devices to better performing states (line 6). If $\overline{R}(t)$ is lower than R_{low}, EASY tries to switch devices to power states which require less energy (line 8). The procedures SPEEDUP() and SLOWDOWN() solve the optimization problems $\mathscr{P}(R_{\text{low}})$ and $\mathscr{P}(R_{\text{high}})$, respectively: they identify a new system configuration $\mathbf{S}'$ such that the estimated response time is less than R_{high}.

By choosing appropriate values for R_{low}, R_{high} and for the monitoring interval Δt, it is possible to achieve different trade-offs between reactiveness and sensitivity to changes in the system response time. Specifically, the duration Δt of the monitoring interval controls the reactiveness of EASY, that is, how promptly system state changes are detected: lower values result in faster reaction times, but also increase the overhead due to the computation of new configurations. The parameters R_{low} and R_{high} control how EASY is sensitive to small deviations of the average response time $\overline{R}(t)$ from the threshold $R_{\max}$.

5 System Modeling

In order to estimate the system response time under different configurations, EASY makes use of a suitable performance model. We model the system as a single-class, closed QN. Devices are represented as queueing centers; requests circulate through the system, joining the queues associated to devices they ask service from. Once service is complete, a request can be forwarded to another device, or may be sent back to the user who submitted it.

In a closed network there is a fixed number of requests which circulate through the system. This represents the fact that each client can have some fixed, maximum number of outstanding requests, and will wait for one of those requests to complete before submitting another one. We allow the number of clients to vary over time, since every Δt time units a new network is built and analyzed from scratch.

QNs are a well studied modeling formalism and provide good tradeoffs between accuracy and efficiency of solution algorithms. In particular, a special class of QNs, called product-form QNs, have a simple closed form expression of the stationary state distribution, so that average performance measures can be computed efficiently [15]. In order to make this paper self-contained, we included additional details on QNs in B.

As already observed, efficiency of the reconfiguration algorithm is of primary importance for EASY, as it must operate on-line and should compute new system configurations as fast as possible. While product-form QNs must satisfy some constraints which somewhat reduce their accuracy in modeling real-world systems (see [15] for details), we argue that they are still accurate enough for guiding the process of identifying new configurations. Also, we remark that the accuracy of a performance model is both related to its expressiveness (how precisely the model can capture the actual system behavior), and to the accuracy of the input parameters which are used to evaluate the model. In EASY, model parameters come from the monitor, which has a rough, and possibly not always up-to-date, vision of the running system. If accurate parameters are not available, there would be little or no justification in using more sophisticated (and possibly slower to analyze) modeling techniques.

The QN model used by EASY contains K service centers, with center k corresponding to device $k \in \mathcal{K}$. The system response time $R(\mathbf{S})$ with configuration $\mathbf{S}$ can be estimated from the average service demand for each device, and the number of requests in the system.

Let $\overline{X}(t)$ be the measured system throughput at time t, $\overline{U}[k](t)$ the measured utilization of device k and $D[k, S[k](t)]$ the service demand of device k in the current state $S[k](t)$. The service demand is the total time spent by a request in device k, and can be defined, according to the Utilization Law [16], as:

$$D[k, S[k](t)] = \frac{\overline{U}[k](t)}{\overline{X}(t)}$$

The service demand $D[k, l]$ for a generic state $l \in \mathcal{L}[k]$ can be estimated as:

$$D[k, l] = \frac{\text{Processing rate of device } k \text{ in state } l}{\text{Processing rate of device } k \text{ in state } S[k](t)} \times \frac{\overline{U}[k](t)}{\overline{X}(t)}$$

which, applying Eq. (1), can be rewritten as:

$$D[k, l] = \frac{RSP[k, S[k](t)]}{RSP[k, l]} \times \frac{\overline{U}[k](t)}{\overline{X}(t)} \tag{3}$$

The number of requests in the system at the current time, $N(t)$, can be computed from the observed system throughput $\overline{X}(t)$ and response time $\overline{R}(t)$ using Little's law [17]:

$$N(t) = \overline{X}(t) \times \overline{R}(t) \tag{4}$$

Using the parameters above, the system response time can be estimated using the function ESTIMATERESPT() shown in Algorithm 2. Given an arbitrary configuration $\mathbf{S}'$, the current configuration $\mathbf{S} = \mathbf{S}(t)$, measured device utilizations $\overline{\mathbf{U}} = \overline{\mathbf{U}}(t)$, system throughput $\overline{X} = \overline{X}(t)$ and response time $\overline{R} = \overline{R}(t)$, Algorithm 2 estimates the system response time $R(\mathbf{S}')$. The estimate is obtained using Balanced System Bounds (BSB) analysis [18], which provides upper and lower bounds (R^+ and R^-, respectively) on the system response time. The system response time is estimated as the average of the upper and lower bound.

We remark that the result computed by function ESTIMATERESPT() is only an estimate of the response time of the QN model. The exact response time of a product-form, closed QN model

Algorithm 2 EstimateRespT$\left(\mathbf{S}', \mathbf{S}, \overline{\mathbf{U}}, \overline{X}, \overline{R}\right) \rightarrow R(\mathbf{S}')$

Require: $\mathbf{S}'$ arbitrary state
Require: $\mathbf{S} = \mathbf{S}(t)$ current state
Require: $\overline{\mathbf{U}} = \overline{\mathbf{U}}(t)$ current device utilizations
Require: $\overline{X} = \overline{X}(t)$ current system throughput
Require: $\overline{R} = \overline{R}(t)$ current system response time
Ensure: $R(\mathbf{S}')$ estimated system response time in state $\mathbf{S}'$
$N := \overline{X} \times \overline{R}$ {Little's Law (4)}
for all $k \in \mathcal{K}$ **do**
$D[k] := D[k, S'[k]] := \dfrac{RSP[k, S[k]]}{RSP[k, S'[k]]} \times \dfrac{\overline{U}[k]}{\overline{X}}$ {Service demands (3)}
$D_{\max} := \max\{D[k] \mid k \in \mathcal{K}\}$
$D_{\text{tot}} := \sum_{k \in \mathcal{K}} D[k]$
$D_{\text{ave}} := D_{\text{tot}}/K$
$R^- := \max\{ND_{\max}, D_{\text{tot}} + (N-1)D_{\text{ave}}\}$ {Lower bound}
$R^+ := D_{\text{tot}} + (N-1)D_{\max}$ {Upper bound}
Return $(R^+ + R^-)/2$

can be computed using the Mean Value Analysis (MVA) algorithm [19]. However, it must be observed that the computational cost for analyzing a network with N requests and K queueing centers using MVA is $O(NK)$; on the other hand, the computational cost of Algorithm 2 is $O(K)$. Not only performance bounds can be computed faster, which is a crucial aspect for EASY: experimental results, which will be described in Section 8, show that this approximation is sufficient to compute almost optimal configurations.

6 Solving the Optimization Problem using Mixed Integer Programming

In this section we describe a formulation of the optimization problem $\mathscr{P}(R_{\max})$ as a MIP problem. The new formulation, which we call $\mathscr{P}_{\text{MIP}}(R_{\max})$, can be solved using known numerical techniques, and many software packages for handling MIP problems are available, both commercial and open source. Therefore, we will use $\mathscr{P}_{\text{MIP}}(R_{\max})$ as a benchmark against which a computationally simpler heuristic, described in Section 7, will be compared.

Let $D[k, l]$ be the service demand of device k when operating in performance state $l \in \mathcal{L}[k]$, as defined in (3). We encode the solution of the optimization problem (2) in a binary matrix $X[k, l]$, such that $X[k, l] = 1$ if and only if device k must be set in state l to get the minimum energy consumption rate. As described in Section 5, we estimate the system response time $R(\mathbf{S}')$ for an arbitrary configuration $\mathbf{S}'$ using BSBs. Then, problem $\mathscr{P}(R_{\max})$ can be formulated as the equivalent MIP problem $\mathscr{P}_{\text{MIP}}(R_{\max})$ shown in Fig. 6.

It should be observed that the problem shown in Fig. 6 is not linear, due to constraints (*) on R_{min} and (**) on $D_{\max}$. Fortunately, we can obtain an equivalent linear problem by replacing (*) with:

$$\begin{aligned} R_{\min} &\geq N D_{\max} \\ R_{\min} &\geq D_{\text{tot}} + (N-1) D_{\text{ave}} \end{aligned}$$

and replacing (**) with the following K constraints:

$$D_{\max} \geq D[k], \quad \text{for all } k \in \mathcal{K}$$

$\mathscr{P}_{\mathrm{MIP}}(R_{\max}) \equiv$

Given:	$\mathcal{K} = \{1, \ldots, K\}$,	Set of devices
	$\mathcal{L}[k] = \{1, \ldots, L[k]\}$,	$k \in \mathcal{K}$; Set of performance states of device k
	$D[k,l]$,	$k \in \mathcal{K}$, $l \in \mathcal{L}[k]$: estimated service demand of device k in performance state l, computed using Eq. (3)
	$EN[k,l]$,	$k \in \mathcal{K}$, $l \in \mathcal{L}[k]$: energy consumption rate of device k in performance state l
	$R_{\max}$,	Maximum system response time
	N	Number of users currently connected to the system
Define:	$X[k,l]$,	$k \in \mathcal{K}$, $l \in \mathcal{L}[k]$: $X[k,l] = 1$ if and only if device k should be set in performance state l, 0 otherwise
Minimize:	$\sum_{k \in \mathcal{K}} \sum_{l \in \mathcal{L}[k]} X[k,l] \times EN[k,l]$,	
		Total energy consumption
Subject to:	$R \leq R_{\max}$,	The system response time should not exceed $R_{\max}$
	$\sum_{l \in \mathcal{L}[k]} X[k,l] = 1$,	$k \in \mathcal{K}$: Each device must be in one performance state only
	$R = (R^+ + R^-)/2$,	The system response time is estimated as the average of upper and lower bounds
(*)	$R^- = \max\left(N D_{\max}, D_{\mathrm{tot}} + (N-1) D_{\mathrm{ave}}\right)$,	
		Lower bound of the response time
	$R^+ = D_{\mathrm{tot}} + (N-1) D_{\max}$,	
		Upper bound of the response time
	$D_{\mathrm{tot}} = \sum_{k \in \mathcal{K}} D[k]$,	
(**)	$D_{\max} = \max_{k \in \mathcal{K}} D[k]$,	
	$D_{\mathrm{ave}} = D_{\mathrm{tot}}/K$,	
	$D[k] = \sum_{l \in \mathcal{L}[k]} X[k,l] \times D[k,l]$,	
		$k \in \mathcal{K}$: $D[k]$ is the service demand of device k with the current configuration encoded by matrix $X[k,l]$

Figure 6. Mixed Integer Programming problem $\mathscr{P}_{\mathrm{MIP}}(R_{\max})$

The resulting MIP problem is linear, and can be solved using existing software packages (e.g., GLPK[3], CPLEX[4], `lp_solve`[5], just to name a few).

It should be observed that the solution of MIP problems can be computationally challenging [20], even for relatively small instances. However, MIP solvers allow the user to specify a maximum allowed computation time: if the optimal solution is not found within the allocated time, the best feasible solution is returned. Thus, the formulation of $\mathscr{P}(R_{\max})$ as a MIP will allow us to use widely used and well tested software solvers which produce high quality (if not optimal) results.

To address the problem of the high computational requirements of solving $\mathscr{P}_{\mathrm{MIP}}(R_{\max})$ as a MIP, we describe in the next section an efficient heuristic algorithm for handling $\mathscr{P}(R_{\max})$.

3. `http://www.gnu.org/software/glpk/`
4. `http://www.ampl.com/`
5. `http://lpsolve.sourceforge.net/`

7 Heuristic solution of the optimization problem

We now describe a simple heuristic solution of the optimization problem (2); the heuristic uses the QN performance model to drive the computation of the new configuration.

The idea is the following: given the initial configuration $\mathbf{S} = \mathbf{S}(t)$ we iteratively compute a new configuration $\mathbf{S}'$ by iteratively switching one device to a higher/lower numbered performance state.

If $R(t) > R_{\text{high}}$, at each iteration we identify the bottleneck device B and switch it to a lower numbered ACPI state, such that the device operates faster; the reconfiguration process stops when either (i) the estimated system response time becomes lower than R_{high}, or (ii) all devices are in state 1, and thus cannot be made any faster.

If $R(t) > R_{\text{low}}$, at each iteration we identify the device with lower utilization and switch it to a higher numbered ACPI state (slower, but requiring less energy). The reconfiguration process stops when no device can be slowed down without making the estimated system response time higher than R_{high},

In both cases above, the reconfiguration process is driven by estimates provided by the QN performance model. As detailed in Sections 7.1 and 7.2 below, the QN model is used both to identify the bottleneck and the device with lowest utilization, and also to estimate the system response time for various configurations. In Section 7.3 we analyze the computational costs of the heuristic, and characterize the results it provides.

7.1 Speeding Up the System

When $R(t) > R_{\text{high}}$, the response time can be improved by identifying and removing the system bottleneck. From queueing theory, it is known that the bottleneck device is the one with highest service demand [16]. However, we must also take into consideration that our goal is to reduce the energy consumption rate: this suggests to give preference to devices which have both a large service demand, and a low energy consumption rate. Specifically, if $\mathbf{S}'$ is the configuration being considered, we select as the bottleneck device k the one for which the ratio $D[k, S'[k]]/EN[k, S'[k]]$ is maximized; in this way we favor devices with high service demand and low energy consumption rate.

The details are shown in Algorithm 3. The procedure SPEEDUP($\mathbf{S}, \mathbf{U}, X, R$) takes as input the current configuration $\mathbf{S}$, the measured device utilizations $\mathbf{U}$ and the system throughput X and response time $R > R_{\text{high}}$. The procedure computes a new configuration $\mathbf{S}'$ which makes the estimated response time less than R_{high}.

The result configuration $\mathbf{S}'$ is computed iteratively, by switching one device at a time to a faster ACPI performance state. At each iteration we compute the set $\mathcal{C}$ of devices which are in a performance state strictly greater than zero; thus, $\mathcal{C}$ is the set of devices which can be made faster. The bottleneck device B is computed as described above, that is the device $B \in \mathcal{C}$ for which the ratio $D[B, S'[B]]/EN[B, S'[B]]$ is maximum (line 5), $S'[B]$ being the performance state of device B in configuration $\mathbf{S}'$. Finally, device B is switched to state $S'[B] - 1$.

Procedure SPEEDUP stops either when (i) the estimated system response time becomes less than R_{max}, or (ii) all devices have been set to state 1 (thus $\mathcal{C}$ becomes empty).

7.2 Slowing Down the System

Algorithm 4 is used to selectively slow down non-bottleneck devices in order to reduce the total energy consumption rate. Again, we use a greedy approach in which a new system configuration $\mathbf{S}'$ is derived from the current one, by slowing down one device at a time. The device U to slow down is selected among a set $\mathcal{C}$ of candidates. Initially, $\mathcal{C}$ simply contains the index of all devices which are not yet in their highest numbered (slower) performance state.

The device U which is selected at each step satisfies the following two properties:

- It is the device whose slowdown produces the minimum increase in the system response time with maximum reduction of energy consumption rate (line 5);

Algorithm 3 SPEEDUP$(\mathbf{S}, \mathbf{U}, X, R) \rightarrow \mathbf{S}'$

Require: $\mathbf{S}$ current system state
Require: $\mathbf{U}$ current device utilizations
Require: X current system throughput
Require: R current system response time, $R > R_{\text{high}}$
Ensure: $\mathbf{S}'$ new system state
1: Compute $D[k, j]$ using (3), for all $k \in \mathcal{K}$, $j \in \mathcal{L}[k]$
2: $\mathbf{S}' := \mathbf{S}$
3: $\mathcal{C} := \{k \in \mathcal{K} \mid S'[k] > 1\}$ {Candidate set}
4: **while** $(\mathcal{C} \neq \emptyset)$ **do**
5: $\quad B := \arg\max_k \left\{ \frac{D[k, S'[k]]}{EN[k, S'[k]]} \;\middle|\; k \in \mathcal{C} \right\}$
6: $\quad S'[B] := S'[B] - 1$ {Speed up device B}
7: $\quad R_{\text{est}} :=$ EstimateRespT$(\mathbf{S}', \mathbf{U}, X, R)$
8: $\quad$ **if** $(R_{\text{est}} < R_{\text{high}})$ **then**
9: $\quad\quad$ **Break** {Complete}
10: $\quad \mathcal{C} := \{k \in \mathcal{K} \mid S'[k] > 1\}$ {Recompute the candidate set}
11: **Return** $\mathbf{S}'$

Algorithm 4 SLOWDOWN$(\mathbf{S}, \mathbf{U}, X, R) \rightarrow \mathbf{S}'$

Require: $\mathbf{S}$ current system state
Require: $U[k]$ current utilization of device k
Require: X current system throughput
Require: R current system response time, $R < R_{\text{low}}$
Ensure: $\mathbf{S}'$ new system state
1: Compute $D[k, j]$ using (3), for all $k \in \mathcal{K}$, $j \in \mathcal{L}[k]$
2: $\mathbf{S}' := \mathbf{S}$
3: $\mathcal{C} := \{k \in \mathcal{K} \mid S'[k] < L[k]\}$ {Devices which can be made slower}
4: **while** $(C \neq \emptyset)$ **do**
5: $\quad U := \arg\min_k \left\{ \frac{D[k, S'[k]+1]}{EN[k, S'[k]+1]} \;\middle|\; k \in C \right\}$
6: $\quad S'[U] := S'[U] + 1$ {Try to slow down device U}
7: $\quad R_{\text{est}} :=$ EstimateRespT$(\mathbf{S}, \mathbf{S}', \mathbf{U}, X, R)$
8: $\quad$ **if** $(R_{\text{est}} > R_{\text{max}})$ **then**
9: $\quad\quad S'[U] := S'[U] - 1$ {Rollback configuration for device U}
10: $\quad\quad \mathcal{C} := \mathcal{C} \setminus \{U\}$ {Device U will no longer be considered}
11: $\quad$ **else if** $(S'[U] = L[U] - 1)$ **then**
12: $\quad\quad \mathcal{C} := \mathcal{C} \setminus \{U\}$
13: **Return** $\mathbf{S}'$

- After setting device U in state $S'[U] + 1$, the estimated system response time is below the threshold R_{high} (lines 6–7).

The procedure terminates when the set $\mathcal{C}$ of candidates becomes empty. Device U is removed from $\mathcal{C}$ if it can not be made any slower (line 11), or if it has been verified that moving device U to a slower performance state violates the constraint $R > R_{\text{high}}$.

7.3 Efficiency and Optimality Considerations

Algorithms 3 and 4 execute a number of iterations in which a single device is reconfigured at a time; in particular, at each iteration one device k is switched from state $S[k]$ to $S[k] + 1$ (in the SLOWDOWN procedure), or from state $S[k]$ to $S[k] - 1$ (in the SPEEDUP procedure).

In order to analyze the solution $\mathbf{S}'$ computed by the procedures above, we introduce some notation. Let $\mathbf{P}$ and $\mathbf{Q}$ two arbitrary system configurations. We say that $\mathbf{P} \prec \mathbf{Q}$ if, for all $k \in \mathcal{K}$, $P[k] \leq Q[k]$, the inequality being strict for at least one value of k.

Using the fact that energy consumption rates are monotonically decreasing (see Section 3),

we conclude that $\mathbf{P} \prec \mathbf{Q}$ implies $E(\mathbf{P}) < E(\mathbf{Q})$. Also, since switching a device k to a higher performance state means reducing its service demand, the following Lemma holds:

Lemma 1 *If $\mathbf{P} \prec \mathbf{Q}$, then $R(\mathbf{P}) > R(\mathbf{Q})$, where $R(\mathbf{S})$ is the estimated system response time computed by analyzing the QN model.*

Lemma 1 derives from known monotonicity properties of QN models that have been proved in [21], and it basically states that if we switch a single device to a upper (lower) performance level, the expected system response time increases (decreases). Note that the Lemma is true also if $R(\mathbf{S})$ is the exact response time of the QN model as computed by the MVA algorithm.

Using the properties above, we can characterize the solutions computed by Algorithms 3 and 4. The invocation of SPEEDUP$(\mathbf{S}, \mathbf{U}, X, R)$ returns a new configuration $\mathbf{S}'$ such that $\mathbf{S}' \prec \mathbf{S}$. From Lemma 1 we have that $R(\mathbf{S}') < R(\mathbf{S})$. Moreover, by inspecting the code we can observe that there exists no configuration $\mathbf{S}'' \succ \mathbf{S}'$ such that $R(\mathbf{S}') < R(\mathbf{S}'') \leq R_{\text{high}}$. Hence, $\mathbf{S}'$ is *Pareto optimal* for problem $\mathscr{P}(R_{\text{high}})$. Similarly, the configuration $\mathbf{S}'$ returned by procedure SLOWDOWN is Pareto optimal for problem $\mathscr{P}(R_{\text{high}})$, in the sense that there exists no configuration $\mathbf{S}'' \succ \mathbf{S}'$ such that $R(\mathbf{S}') < R(\mathbf{S}'') \leq R_{\text{high}}$.

From the computational point of view, the execution time of Algorithms 3 and 4 can be computed as the product of the number of iterations and the cost of each iteration. The number of iterations is $O\left(\sum_{k \in \mathcal{K}} |S'[k] - S[k]|\right)$, and each iteration is dominated by the cost of estimating the system response time using Algorithm 2, that is $O(K)$. Hence, the cost of procedure SPEEDUP and SLOWDOWN is $O\left(K \times \sum_{k \in \mathcal{K}} |S'[k] - S[k]|\right)$, which in the worst case is $O\left(K \times \sum_{k \in \mathcal{K}} L[k]\right)$. Note that the total number of possible configurations is $O\left(\prod_{k \in \mathcal{K}} L[k]\right)$, hence the complete exploration of the solution space using bounds to estimate the system response time would cost $O\left(K \times \prod_{k \in \mathcal{K}} L[k]\right)$. results.tex

8 Numerical Evaluation

In this section we assess the effectiveness of EASY by means of a set of synthetic test cases which have been evaluated numerically.

We consider a system with K devices, each supporting the same number L of ACPI performance states; this simplification, which is not required by EASY, allows us to limit the number of simulation parameters. We experiment with all combinations of K and L with $K \in \{10, 20, 30, 50\}$ and $L \in \{2, 4, 6\}$. The relative speed $RSP[k, j]$ of device k in performance state $j \in \{1, \ldots, L\}$ is defined as a linear function of j, with minimum value $RSP[k, L] = 0.3$ and maximum $RSP[k, 1] = 1$. Therefore

$$RSP[k, j] = 1.0 - \frac{j - 1}{L - 1} \times 0.7$$

The energy consumption rate $EN[k, j]$ has been normalized in the range $[0.4, 1]$ in order to get absolute results which are independent from the actual energy consumption rates of a specific processor. $EN[k, j]$ has been defined as:

$$EN[k, j] = 1.0 - \frac{j - 1}{L - 1} \times 0.6$$

which is a linear function of j with minimum $EN[k, 1] = 1$ and minimum $EN[k, L] = 0.4$. Linear dependence of the energy consumption rate and relative speed is consistent with data from actual processors (see Tables 1–3).

We performed a time-stepped simulation with $T = 300$ steps. Algorithm 1 is executed every $\Delta t = 5$ time steps. We simulated a variable workload by changing the number of users $N(t)$ at step t. In order to use realistic data, we monitored a real large scale system by collecting the total number of online players connected to the RuneScape MMOG[6] in May 2011. RuneScape

6. http://www.runescape.com/

is a Fantasy MMOG where players can travel across the fictional medieval realm. Players use an ordinary Web browser to connect to one of the available RuneScape servers; the servers are located in different world regions, so that communication latency can be minimized. During peak hours, more than 200.000 players are connected to the system (the load is split across the regional servers); this number becomes as low as about 110.000 players during off-peak hours. We extracted a subset of the data spanning 4 days; the data have been downsampled in order to fit four days in $T = 300$ time steps. Finally, the sampled data have been rescaled so that the maximum number of users is about 100 (a workload which can be reasonably handled by the system for all values of K and L).

The service demand $D[k](t)$ for device k at simulation step t has been generated by drawing uniformly distributed random values in the range $[0.2, 1.0]$; this ensured that service demands are not constant over time. We smoothed the service demand data by computing moving averages for each device, using a window size of 10 steps.

For each experiment the "hard" threshold $R_{\max}$is defined as follows. Let $t_{\max} = \arg\max\{N(t) \mid t = 1, \ldots, T\}$ be the time step at which there is the maximum number of concurrent users. We use the MVA algorithm to compute the system response time $\overline{R}(t_{\max})$ at step $t_{\max}$, using the number of requests $N(t_{\max})$ as above, and with all devices set in performance state L (slower). We then define $R_{\max} = 0.8 \times \overline{R}(t_{\max})$, $R_{\text{high}} = 0.9 \times R_{\max}$ and $R_{\text{low}} = 0.8 \times R_{\max}$. The above definition of $R_{\max}$ ensures that, at each time step $t \in \{1, \ldots, T\}$, there always exists a configuration $\mathbf{S}(t)$ which solves the optimization problem $\mathscr{P}(R_{\max})$.

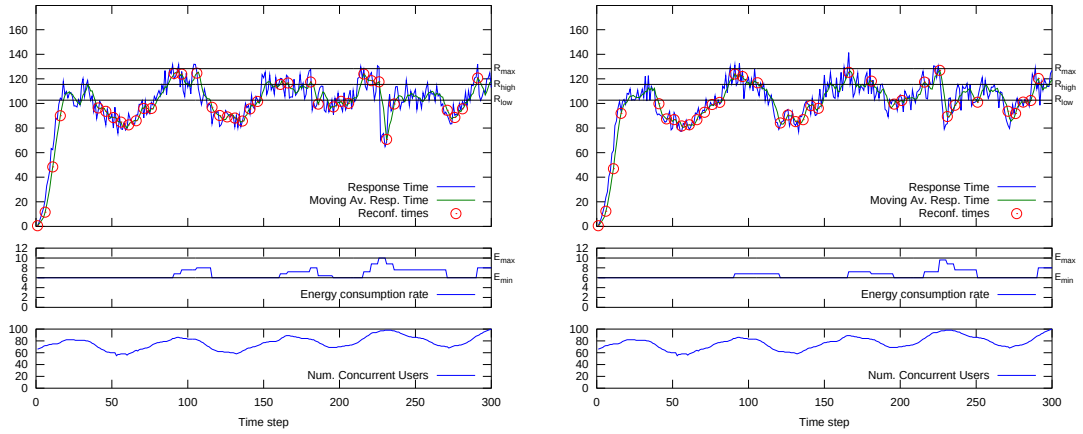
EASY is executed on-line, that is it finds a new configuration at time step t by considering only the configuration at the previous step $t - 1$ and the number of currently active requests N_t. The observed system response time $\hat{R}(t)$ at time t is computed as follows: first, we compute the system response time from the QN model using MVA; then, we multiply this value by a random number uniformly distributed in $[0.9, 1.1]$. This is used to simulate the fact that, in practice, the QN model will not produce the correct response time estimates.

We implemented Algorithms 3 and 4 in GNU Octave [22], an interpreted language for numerical computations. We implemented the main control loop (Algorithm 1) and procedures SPEEDUP() and SLOWDOWN() (Algorithms 3 and 4). We also compared the results with those obtained by replacing SPEEDUP() and SLOWDOWN() with the computation of exact solutions of the MIP problem $\mathscr{P}_{\text{MIP}}(R_{\text{high}})$ (note that Algorithms 3 and 4 return a configuration $\mathbf{S}'$ such that $R(\mathbf{S}') \leq R_{\text{high}}$). The MIP problem is handled by GLPK, with the additional constraint that if no optimal solution is found within 60 seconds, the best feasible solution is returned. All tests have been performed on an AMD Athlon 64 X2 3800+ Dual Core Processor with 4 GB of RAM running Linux 2.6.32; we used GNU Octave version 3.2.3 and GLPK version 4.45.

For each combination of K and L, we performed 10 independent simulation runs. We then computed average performance measures and 90% confidence intervals.

Figure 7 shows an example of the results for a system with $K = 10$ components operating at $L = 2$ quality levels. Figure 7(a) shows a run where EASY uses the heuristic algorithms described in Section 7, while Fig. 7(b) shows a run where EASY computes an exact solution of the MIP problem with the help of GLPK. The top part of the plot shows the observed system response time $\overline{R}(t)$ at step t (thick line), together with the moving average (thin line); the latter is used by EASY to decide when a reconfiguration should be triggered, according to Algorithm 1. Reconfiguration points are also shown as small circles: these are the times in which either procedure SPEEDUP() or SLOWDOWN() have been invoked. The middle part of Fig. 7 shows the energy consumption rate $E(\mathbf{S}(t))$; horizontal lines show the minimum energy consumption rate, corresponding to the configuration in which all devices are in performance state L, and the maximum energy consumption rate, corresponding to the configuration in which all devices are in the fastest performance state 1. Finally, the bottom part of Fig. 7 shows the number of users $N(t)$ at time step t.

We now describe the performance metrics which have been considered in our study, and discuss the results obtained from the simulation runs. For completeness, the raw data are reported in Tables 4 and 5. Table 6 allows quick comparison between the heuristic algorithm and



(a) Heuristic (b) MIP

Figure 7. Results for a single simulation run with $K = 10$ devices and $L = 2$ performance states.

the MIP solver; for each row, we underline the best result whose confidence interval does not overlap with the alternative.

Number of SLA violations This is the number of time steps in which the response time constraints $\overline{R}(t) \leq R_{\max}$ has been violated. This is a *lower is better* metric, which means that lower values are preferred.

Figure 8 shows the results obtained with the heuristic and the MIP solver. The height of histograms represents the average of 10 independent simulation runs, and the error bars denote 90% confidence intervals. We observe that, in most cases, the number of violations produced by the heuristic and by the MIP solver are compatible at this confidence level; this means that the configurations produced by finding an optimal solution of the MIP problem are not significantly better than those produced by the heuristic. However, if we look at the average values alone (height of histograms), we observe that the heuristic produces marginally less SLA violations than the MIP solver. This may seem counter intuitive, as solutions of the MIP problem are expected to be the "best" possible configurations. The observed results can be easily explained by observing that the configurations produced by the heuristic require more energy than those produced by the MIP solver. Thus, the heuristic essentially trades more energy for better (faster) configurations which yield a marginally lower average number of SLA violations.

Total response time overflow ΔR The total response time overflow gives a measure of "how much" the SLA constraint has been violated. ΔR is defined as the sum, over all times t where $\overline{R}(t) > R_{\max}$, of the difference $\overline{R}(t) - R_{\max}$. In other words, ΔR is the area which lies above the line of the response time constraint $R_{\max}$ and below the curve of measured response time $\overline{R}(\mathbf{S}(t))$ in Fig. 7.

We show in Fig. 9 the comparison of the response time overflow produced by the heuristic and the MIP solver. Again, results in both cases are not statistically different, although average values seem to indicate that the heuristic is marginally better than the MIP solver. As for the average number of SLA violations, this can be explained by considering that configurations produced by the heuristic have higher energy consumption rate, so the heuristic is trading energy for lower response times.

Total energy consumption E_{tot} The total energy consumption is defined as the sum of consumption rates $E(\mathbf{S}(t))$ over all time steps $t = 1, \ldots, T$, where $\mathbf{S}(t)$ is the configuration selected by EASY at step t.

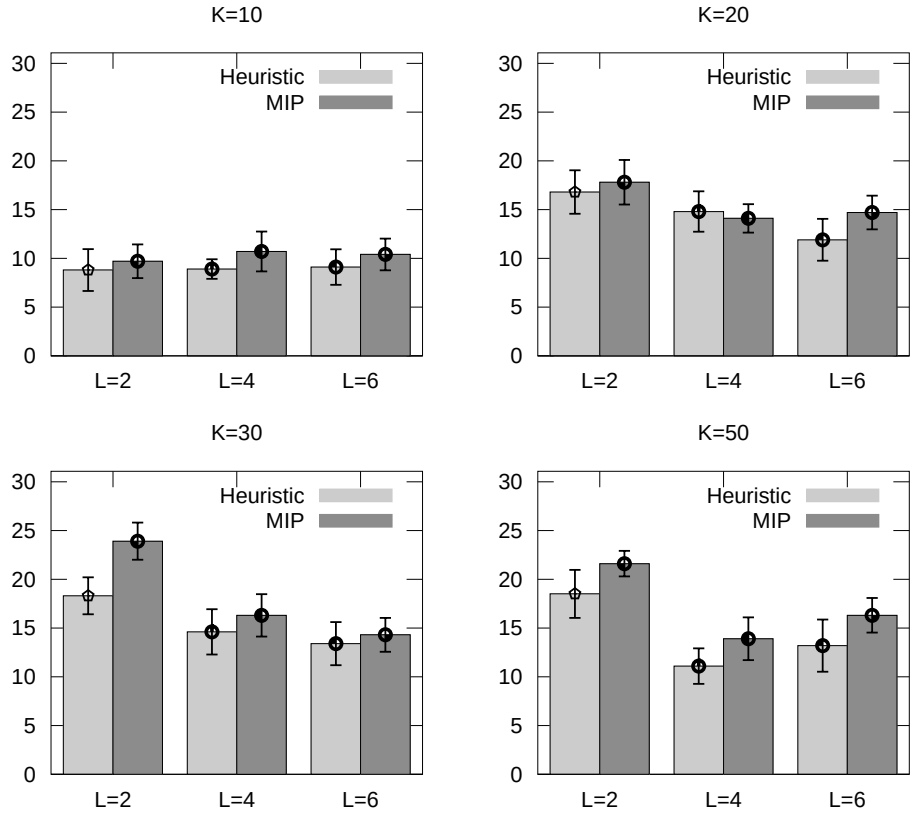


Figure 8. Number of SLA violations (lower is better)

Results are shown in Fig. 10; referring to the raw data on Table 6, we observe that the MIP solver produces configurations with lower energy consumption.

Execution time We measured the total execution time of each simulation run; it must be observed that the results should not be taken as absolute performance measures, since EASY has been implemented using an interpreted language which is far less efficient than a compiled one. Furthermore, different MIP solvers may exhibit different convergence speed on the same problem instance, or may even fail to converge in reasonable time scales. Therefore, the results shown here should be considered as a qualitative measure of efficiency of the heuristic with respect to the solution of the MIP problem.

As can be seen from Fig. 11, for small values of K the heuristic described by Algorithms 3 and 4 requires almost the same time as the exact solution of the MIP problem using GLPK. However, as the problem size increases, GLPK is much slower than the heuristic, the difference being more than an order of magnitude for $K = 50$. We remark that we forced GLPK to produce a solution within 60 seconds; for large values of K (e.g., $K = 50$), the likelihood of GLPK not being able to find a solution within the allocated time increases.

Total energy overhead E_{ovr} This parameter is used to estimate the efficiency of EASY by measuring the fraction of energy, above the minimum consumption rate, which is used by the system. Formally, let $E_{\min}$ be the minimum energy consumption rate when all devices are in state L, and let $E_{\max}$ be the maximum energy consumption rate when all devices are in state 1. Let $E(\mathbf{S}(t))$ be the energy consumption rate at time t, under the configuration $\mathbf{S}(t)$ produced by EASY. Then,

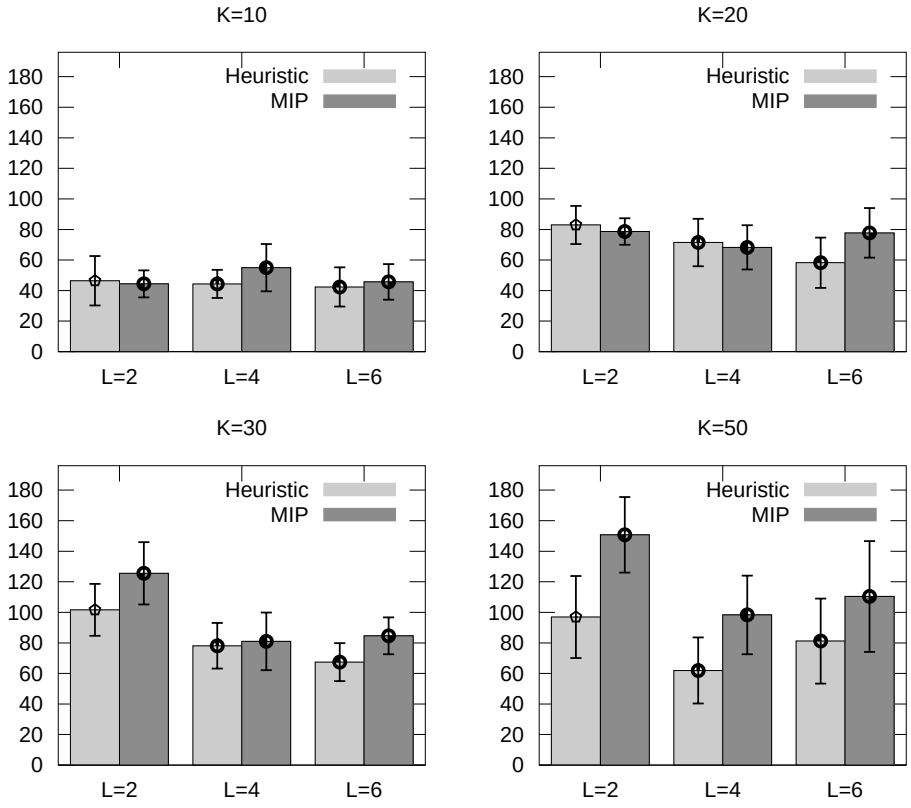


Figure 9. Response time overflow ΔR (lower is better)

the energy overhead is defined as:

$$E_{\text{ovr}} = \frac{\sum_{t=1}^{T} (E(\mathbf{S}(t)) - E_{\min})}{T \times (E_{\max} - E_{\min})}$$

By definition, the energy overhead is a number in the range $[0, 1]$. The value $E_{\text{ovr}} = 1$ corresponds to the situation in which all devices are always set in state 0; the value $E_{\text{ovr}} = 0$ corresponds to the situation in which all devices are in state L. Therefore, the energy overhead shows how efficiently the available energy consumption range between $E_{\min}$ and $E_{\max}$ is being used.

The values of E_{ovr} for all experiments are reported in Tables 4 and 5. From the results we can make an important observation: increasing the number of ACPI performance states L yields a lower value of E_{ovr}. In fact, increasing the number of performance states allows EASY to lower the energy consumption of each device to the minimum level allowing it to deliver "just enough" processing power to accommodate its service demand.

9 Related Work

In the last years, as outlined in [23], the topic of adaptive computing systems has been studied in several communities and from different perspectives. The *autonomic computing* framework, for example, is a notable example of general approach to the design of such systems [24, 25]. Our work lies in the area of models for self-adaptation of systems able to guarantee the fulfillment of QoS requirements under variable workload and reducing the energy consumption. Therefore, hereafter, we focus on works appeared in the literature dealing with (i) *dynamic power management* and (ii) *trade-off between energy consumption and performance*.

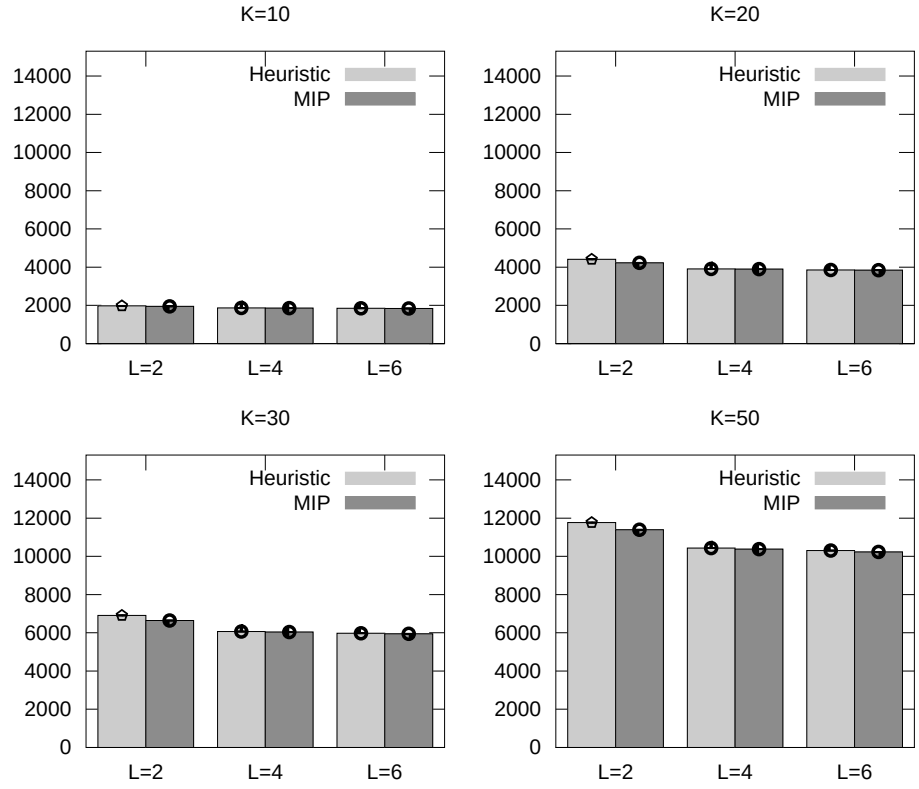


Figure 10. Total energy consumption E_{tot} (lower is better)

Table 4. Results for the heuristic approach

K	L	SLA Violations	ΔR	E_{tot}	E_{ovr}	Exec. Time
10	2	8.90 ± 2.21	76.29 ± 28.12	1454.70 ± 18.88	0.14	4.43 ± 0.03
10	4	8.60 ± 1.03	71.42 ± 16.65	1292.90 ± 3.10	0.05	4.73 ± 0.03
10	6	8.90 ± 2.45	67.75 ± 24.58	1256.88 ± 3.47	0.03	5.03 ± 0.03
20	2	16.70 ± 2.49	137.15 ± 26.16	3569.40 ± 39.91	0.32	5.35 ± 0.04
20	4	17.00 ± 1.83	140.08 ± 24.86	2822.20 ± 11.28	0.12	5.92 ± 0.06
20	6	14.30 ± 2.24	116.35 ± 26.47	2667.84 ± 9.26	0.07	6.62 ± 0.07
30	2	25.10 ± 2.40	267.09 ± 48.80	5627.70 ± 27.37	0.38	6.30 ± 0.12
30	4	16.80 ± 1.95	172.47 ± 33.57	4386.00 ± 9.39	0.15	7.52 ± 0.15
30	6	17.60 ± 2.12	159.38 ± 30.82	4124.64 ± 5.11	0.10	8.27 ± 0.17
50	2	26.60 ± 1.97	246.11 ± 31.73	9517.80 ± 51.00	0.39	7.61 ± 0.11
50	4	16.30 ± 1.50	140.97 ± 27.14	7484.70 ± 10.14	0.16	9.98 ± 0.18
50	6	15.20 ± 1.54	182.25 ± 43.83	7064.34 ± 7.46	0.12	12.12 ± 0.25

ICT-based *dynamic power management* (DPM) aim is to reconfigure system components or entire systems so that they operate with minimal energy consumption. To this end, several techniques for resource allocation under workload fluctuations have been studied and applied [26], including techniques for admission control and load balancing. *Admission control* is a system overload protection mechanism that rejects requests under peak conditions in order to provide performance guarantees to the running requests [27, 26]. *Load balancing* aims to partition incoming requests among multiple instances of the same application in order to lead to a better utilization of system resources or to provide performance guarantees (e.g., [28, 29]). Integrated

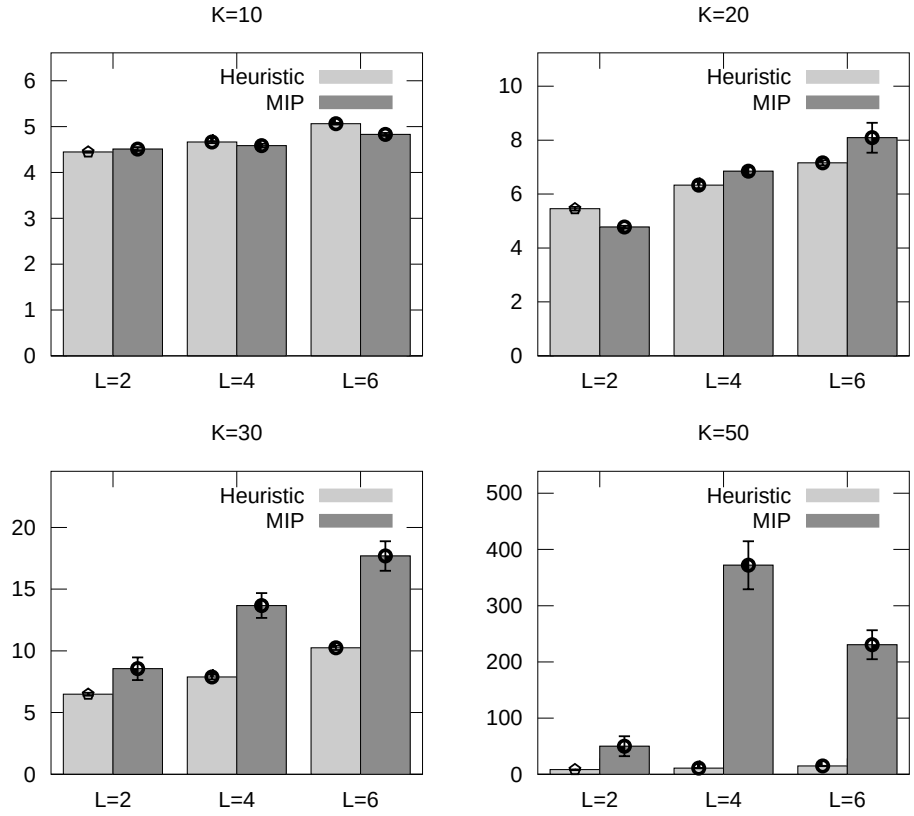


Figure 11. Execution time (lower is better). Note that each subfigure has different scale.

Table 5. Results for the approach involving exact solution of the MIP problem

K	L	SLA Violations	ΔR	E_{tot}	E_{ovr}	Exec. Time
10	2	9.70 ± 1.59	73.67 ± 14.63	1418.70 ± 13.87	0.12	4.50 ± 0.04
10	4	9.50 ± 2.01	79.50 ± 28.47	1277.20 ± 2.76	0.04	4.64 ± 0.04
10	6	9.70 ± 1.37	64.31 ± 15.39	1246.68 ± 1.89	0.03	4.81 ± 0.04
20	2	16.50 ± 1.67	115.84 ± 14.10	3289.80 ± 31.09	0.25	4.80 ± 0.06
20	4	17.60 ± 2.32	131.35 ± 24.20	2740.80 ± 13.71	0.09	5.90 ± 0.09
20	6	16.20 ± 1.54	145.47 ± 21.15	2627.82 ± 5.75	0.06	8.32 ± 0.36
30	2	29.30 ± 3.10	244.69 ± 19.78	5218.20 ± 59.86	0.30	12.01 ± 0.21
30	4	20.30 ± 1.52	173.08 ± 39.12	4278.10 ± 10.03	0.13	54.86 ± 15.98
30	6	16.30 ± 1.57	186.43 ± 29.05	4089.18 ± 11.03	0.09	40.47 ± 9.65
50	2	33.30 ± 1.91	310.63 ± 41.28	8827.80 ± 38.44	0.31	195.58 ± 19.93
50	4	17.90 ± 2.45	208.40 ± 48.19	7336.50 ± 17.71	0.15	339.11 ± 31.76
50	6	15.80 ± 2.04	191.60 ± 59.64	6998.70 ± 7.18	0.11	146.01 ± 36.48

frameworks such as IBMs WebSphere Application Server and the Tivoli suite combine admission control and load balancing in cluster or data-centre environments [30].

Other DPM solutions are based on system reconfiguration steps that include powering off components while idle, and switching them to low-power operating modes when underutilised (see for example, [31, 32, 33] and [34] for a recent survey). Recent surveys [34, 3, 35] indicate that most of the DPM techniques developed so far are targeted at individual components (like processors or disk drives) or at specific systems (like clusters or data centers), see for example [36,

Table 6. Comparison of heuristic and exact solution of the MIP problem

		MIP				Heuristic			
K	L	SLA Viol.	ΔR	E_{tot}	Exec. Time	SLA Viol.	ΔR	E_{tot}	Exec. Time
10	2	9.70 ± 1.59	73.67 ± 14.63	1418.70 ± 13.87	4.50 ± 0.04	8.90 ± 2.21	76.29 ± 28.12	1454.70 ± 18.88	4.43 ± 0.03
10	4	9.50 ± 2.01	79.50 ± 28.47	1277.20 ± 2.76	4.64 ± 0.04	8.60 ± 1.03	71.42 ± 16.65	1292.90 ± 3.10	4.73 ± 0.03
10	6	9.70 ± 1.37	64.31 ± 15.39	1246.68 ± 1.89	4.81 ± 0.04	8.90 ± 2.45	67.75 ± 24.58	1256.88 ± 3.47	5.03 ± 0.03
20	2	16.50 ± 1.67	115.84 ± 14.10	3289.80 ± 31.09	4.80 ± 0.06	16.70 ± 2.49	137.15 ± 26.16	3569.40 ± 39.91	5.35 ± 0.04
20	4	17.60 ± 2.32	131.35 ± 24.20	2740.80 ± 13.71	5.90 ± 0.09	17.00 ± 1.83	140.08 ± 24.86	2822.20 ± 11.28	5.92 ± 0.06
20	6	16.20 ± 1.54	145.47 ± 21.15	2627.82 ± 5.75	8.32 ± 0.36	14.30 ± 2.24	116.35 ± 26.47	2667.84 ± 9.26	6.62 ± 0.07
30	2	29.30 ± 3.10	244.69 ± 19.78	5218.20 ± 59.86	12.01 ± 0.21	25.10 ± 2.40	267.09 ± 48.80	5627.70 ± 27.37	6.30 ± 0.12
30	4	20.30 ± 1.52	173.08 ± 39.12	4278.10 ± 10.03	54.86 ± 15.98	16.80 ± 1.95	172.47 ± 33.57	4386.00 ± 9.39	7.52 ± 0.15
30	6	16.30 ± 1.57	186.43 ± 29.05	4089.18 ± 11.03	40.47 ± 9.65	17.60 ± 2.12	159.38 ± 30.82	4124.64 ± 5.11	8.27 ± 0.17
50	2	33.30 ± 1.91	310.63 ± 41.28	8827.80 ± 38.44	195.58 ± 19.93	26.60 ± 1.97	246.11 ± 31.73	9517.80 ± 51.00	7.61 ± 0.11
50	4	17.90 ± 2.45	208.40 ± 48.19	7336.50 ± 17.71	339.11 ± 31.76	16.30 ± 1.50	140.97 ± 27.14	7484.70 ± 10.14	9.98 ± 0.18
50	6	15.80 ± 2.04	191.60 ± 59.64	6998.70 ± 7.18	146.01 ± 36.48	15.20 ± 1.54	182.25 ± 43.83	7064.34 ± 7.46	12.12 ± 0.25

37, 38].

Trade-off between energy consumption and performance has been widely investigated in hardware design and network communities, as well as in controlling battery-powered devices since energy savings in these kind of devices have been an intrinsic concern since their creation. However, the interest in trade-offs between performance and energy in hosting centers is much more recent. In [39, 3] it is recognized the importance of the problem of energy wastes. They treat the problem from different points of view, such as hardware devices consumption, operating system or software applications. They sum up previous efforts in the field, raise current problems and devise ways to reduce the energy consumption.

Authors in [31, 4, 40, 41] have addressed the trade-off between performance and energy savings. In [40] authors propose a framework for multi-service hosting platforms that allows the management of reallocation of the correct amount of resources for each service while satisfying the performance requirements. The work in [41] extends the previous one by considering energy consumption constraints and situations where the system is under illegitimate users requests. The work in [4] deals with power consumption and SLA meeting using techniques like switch off servers and modify their working frequency. To manage the processing capacity assigned to each hosted application, they propose methods based on queuing theory, feedback control and a hybrid mechanisms.

With respect to these works, EASY lies at the confluence of these research lines fostering the usage of models at runtime to drive the QoS-based system adaptation under variable workload and reducing the energy consumption. To this end, EASY uses an efficient modeling and analysis technique that can then be used at runtime without undermining the system behavior and its overall performance.

10 Conclusions

In this paper we proposed EASY, a framework for reducing the energy consumption rate of large-scale systems with QoS constraints. EASY can be applied to any system made of a set of ACPI-capable devices. The goal of EASY is to set each device to an appropriate ACPI performance state such that (i) the overall system response time is kept below a given threshold, and (ii) the total energy consumption rate is minimized. We formulate the energy-minimization problem as a MIP problem; using a general-purpose MIP solver we can compute exact solutions of the optimization problem, but unfortunately this approach is infeasible even for moderately complex systems due to the large amount of time required to find the solution. Therefore, we propose an heuristic solution which uses a QN performance model to quickly estimate the response time of a subset of system configurations. We validated the heuristic through numerical experiments, and we observed that the solutions provided by the heuristic are comparable to those obtained by solving the MIP problem; moreover, the heuristic is much faster (more than an order of magnitude) for a system with $K = 50$ devices.

This approach can be extended along different directions. It would be certainly useful adding the possibility to handle non-operating device states (e.g., suspended or powered off) in order to further reduce the power consumption during off peak periods. As described in the introduction, this would pose a set of nontrivial challenges. From one side it would be necessary to migrate data or whole applications away from devices which are being shut down. From the other side it might be appropriate to adopt forecasting techniques in order to predict workload fluctuations ahead of time, in order to have the time to bring machines up or down as requested. Combining EASY with existing energy-conservation approaches based on the use of active/suspended states (e.g., [33]) could be fruitful.

Another direction in which EASY can be extended is that of implementing the control loop shown in Fig. 4 in a totally decentralized way, in order to avoid the possibility that EASY becomes a performance bottleneck or a single point of failure.

References

[1] J. G. Koomey, Estimating total power consumption by servers in the u.s. and the world (Feb. 5 2007).
URL http://sites.amd.com/de/Documents/svrpwrusecompletefinal.pdf

[2] R. Das, J. O. Kephart, C. Lefurgy, G. Tesauro, D. W. Levine, H. Chan, Autonomic multi-agent management of power and performance in data centers, in: AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2008, pp. 107–114.

[3] P. Ranganathan, Recipe for efficiency: principles of power-aware computing, Commun. ACM 53 (4) (2010) 60–67.

[4] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, N. Gautam, Managing server energy and operational costs in hosting centers, SIGMETRICS Perform. Eval. Rev. 33 (1) (2005) 303–314. doi:10.1145/1071690.1064253.

[5] L. A. Barroso, U. Hölzle, The case for energy-proportional computing, Computer 40 (12) (2007) 33–37. doi:10.1109/MC.2007.443.

[6] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, T. F. Wenisch, Power management of online data-intensive services.

[7] Advanced configuration and power interface specification, revision 4.0a, Available at http://www.acpi.info/ (Apr. 5 2010).

[8] L. Brown, A. Keshavamurthy, D. Shaohua Li, R. Moore, V. Pallipadi, L. Yu, ACPI in linux: Architecture, advances and challenges, in: Proceedings of the Linux Symposium–Volume One, Ottawa, Ontario, Canada, 2004.
URL http://www.linuxsymposium.org/2005/

[9] J. M. Rabaey, A. Chandrakasan, B. Nikolic, Digital Integrated Circuits–A Design Perspective, Prentice Hall, 2003, 2nd Edition.

[10] Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor, White Paper, available at ftp://download.intel.com/design/network/papers/30117401.pdf, Accessed on 2010/10/24 (Mar. 2004).

[11] AMD PowerNow! Technology, http://amd.com/us/products/technologies/amd-powernow-technology/Pages/amd-powernow-technology.aspx, Accessed on 2010/10/24.

[12] M. Broyles, C. Francois, A. Geissler, M. Hollinger, T. Rosedahl, G. Silva, J. V. Heuklon, B. Veale, IBM EnergyScale for POWER7 Processor-Based Systems, White Paper, http://www-03.ibm.com/systems/power/hardware/whitepapers/energyscale7.html, Accessed on 2010/10/25 (Aug. 2010).

[13] AMD Cool'n'Quiet Technology, http://www.amd.com/us/products/technologies/cool-n-quiet/Pages/cool-n-quiet.aspx, Accessed on 2010/10/24.

[14] AMD Opteron Processor Power and Thermal Data Sheet, Publication # 30417, Revision 3.11, available at http://support.amd.com/us/Processor_TechDocs/30417.pdf, Accessed on 2011/06/16 (May 2006).

[15] S. Balsamo, Product form queueing networks, in: G. Haring, C. Lindemann, M. Reiser (Eds.), Performance Evaluation: Origins and Directions, Vol. 1769 of Lecture Notes in Computer Science, Springer Berlin/Heidelberg, 2000, pp. 377–401. doi:10.1007/3-540-46506-5_16.

[16] E. D. Lazowska, J. Zahorjan, G. S. Graham, K. C. Sevcik, Quantitative System Performance: Computer System Analysis Using Queueig Network Models, Prentice-Hall, 1984.

[17] J. D. C. Little, A proof for the queuing formula: $L = \lambda W$, Operations Research 9 (3) (1961) 383–387. doi:10.2307/167570.

[18] J. Zahorjan, K. C. Sevcick, D. L. Eager, B. I. Galler, Balanced job bound analysis of queueing networks, Comm. ACM 25 (2) (1982) 134–141.

[19] M. Reiser, S. S. Lavenberg, Mean-value analysis of closed multichain queuing networks, Journal of the ACM 27 (2) (1980) 313–322.

[20] S. G. Nash, A. Sofer, Linear and Nonlinear Programming, McGraw-Hill, New York, 1996.

[21] J. Lüthi, G. Haring, Mean value analysis for queueing network models with intervals as input parameters, Performance Evaluation 32 (3) (1998) 185–215. doi:10.1016/S0166-5316(97)00021-7.

[22] J. W. Eaton, GNU Octave Manual, Network Theory Limited, 2002.

[23] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee (Eds.), Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar], Vol. 5525 of Lecture Notes in Computer Science, Springer, 2009. doi:10.1007/978-3-642-02161-9.

[24] J. O. Kephart, D. M. Chess, The vision of autonomic computing, IEEE Computer 36 (1) (2003) 41–50.

[25] M. C. Huebscher, J. A. McCann, A survey of autonomic computing–degrees, models, and applications, ACM Comput. Surv. 40 (3).

[26] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, T. Wood, Agile dynamic provisioning of multi-tier internet applications, ACM Trans. Auton. Adapt. Syst. 3 (1) (2008) 1–39. doi:10.1145/1342171.1342172.

[27] H. Feng, Z. Liu, C. H. Xia, L. Zhang, Load shedding and distributed resource control of stream processing networks, Perform. Eval. 64 (2007) 1102–1120. doi:10.1016/j.peva.2007.06.023.

[28] R. F. Berry, Trends, challenges and opportunities for performance engineering with modern business software, IEE Proceedings - Software 150 (4) (2003) 223–229.

[29] E. di Nitto, D. J. Dubois, R. Mirandola, On exploiting decentralized bio-inspired self-organization algorithms to develop real systems, in: Proceedings of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, IEEE Computer Society, Washington, DC, USA, 2009, pp. 68–75. doi:10.1109/SEAMS.2009.5069075.

[30] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, A. Tantawi, Analytic modeling of multitier internet applications, ACM Trans. Web 1. doi:10.1145/1232722.1232724.

[31] E. N. Elnozahy, M. Kistler, R. Rajamony, Energy conservation policies for web servers, in: USENIX Symposium on Internet Technologies and Systems, 2003.

[32] R. Calinescu, M. Z. Kwiatkowska, Using quantitative analysis to implement autonomic it systems, in: ICSE, IEEE, 2009, pp. 100–110.

[33] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, R. Katz, Napsac: design and implementation of a power-proportional web cluster, SIGCOMM Comput. Commun. Rev. 41 102–108. doi:http://doi.acm.org/10.1145/1925861.1925878.
URL http://doi.acm.org/10.1145/1925861.1925878

[34] Y. Liu, H. Zhu, A survey of the research on power management techniques for high-performance systems, Software: Practice and Experience 40 (11) (2010) 943–964. doi:10.1002/spe.952.

[35] J. B. Carter, K. Rajamani, Designing energy-efficient servers and data centers, IEEE Computer 43 (7) (2010) 76–78.

[36] D. Barbagallo, E. Di Nitto, D. J. Dubois, R. Mirandola, A bio-inspired algorithm for energy optimization in a self-organizing data center, in: Proceedings of the First international conference on Self-organizing architectures, SOAR'09, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 127–151.

[37] L. Bertini, J. C. Leite, D. Moss, Power optimization for dynamic configuration in heterogeneous web server clusters, Journal of Systems and Software 83 (4) (2010) 585 – 598. doi:10.1016/j.jss.2009.10.040.

[38] M. Marzolla, O. Babaoglu, F. Panzieri, Server consolidation in clouds through gossiping, Technical Report UBLCS-2011-01, Department of Computer Science, University of Bologna, Italy (Jan. 2011).
URL http://www.cs.unibo.it/pub/TR/UBLCS/ABSTRACTS/2011.bib?ncstrl.cabernet//BOLOGNA-UBLCS-2011-01

[39] R. Bianchini, R. Rajamony, Power and energy management for server systems, Computer 37 (11) (2004) 68 – 76. doi:10.1109/MC.2004.217.

[40] B. Abrahao, V. Almeida, J. Almeida, A. Zhang, D. Beyer, F. Safai, Self-adaptive sla-driven capacity management for internet services, 2006, pp. 557 –568. doi:10.1109/NOMS.2006.1687584.

[41] I. Cunha, I. Viana, J. Palotti, J. Almeida, V. Almeida, Analyzing security and energy tradeoffs in autonomic capacity management, 2008, pp. 302 –309. doi:10.1109/NOMS.2008.4575148.

[42] G. Bolch, S. Greiner, H. de Meer, K. Trivedi, Queuing Network and Markov Chains, John Wiley, 1998.

[43] F. Baskett, K. M. Chandy, R. R. Muntz, F. G. Palacios, Open, closed, and mixed networks of queues with different classes of customers, J. ACM 22 (2) (1975) 248–260.

[44] G. Casale, R. Muntz, G. Serazzi, Geometric bounds: A noniterative analysis technique for closed queueing networks, IEEE Trans. Comput. 57 (6) (2008) 780–794.

[45] R. Jain, The Art of Computer Systems Performance Analysis–Techniques for Experimental Design, Measurement, Simulation, and Modeling, Wiley-Interscience, 1991.

[46] A. Makhorin, Modeling Language GNU MathProg–Language Reference for GLPK version 4.45, draft Edition, available at http://www.gnu.org/software/glpk/glpk.html (Dec. 2010).

[47] R. Fourer, D. M. Gay, B. W. Kernighan, A modeling language for mathematical programming, MANAGEMENT SCIENCE 36 (5) (1990) 519–554. doi:10.1287/mnsc.36.5.519.

[48] A. Makhorin, GNU Linear Programming Kit–Reference Manual for GLPK version 4.45, draft Edition, available at http://www.gnu.org/software/glpk/glpk.html (Dec. 2010).

A Notation

In Table 7 we summarize the main notation adopted throughout the paper.

$\mathcal{K}$	Index set of devices, $\mathcal{K} = \{1, \ldots, K\}$
$\mathcal{L}[k]$	Index set of performance states of device k, $\mathcal{L}[k] = \{1, \ldots, L[k]\}$
$RSP[k, l]$	Relative speed of device k in state $l \in \mathcal{L}[k]$ with respect to the same device in state 0 (fastest)
$EN[k, l]$	Energy consumption rate of device k in state $l \in \mathcal{L}[k]$
$\mathbf{S}, \mathbf{S}'$	System configurations; $\mathbf{S} = (S[1], \ldots, S[K])$, where $S[k] \in \mathcal{L}[k]$ is the current performance state of device k
$\mathbf{S}(t)$	System configuration at time t
$E(\mathbf{S})$	Total energy consumption rate of the system in state $\mathbf{S}$
$\overline{R}(t)$	Measured system response time at time t
$\overline{X}(t)$	Measured system throughput at time t
$\overline{U}[k](t)$	Measured utilization of device $k \in \mathcal{K}$ at time t
$R(\mathbf{S})$	Estimated system response time under configuration $\mathbf{S}$
$D[k]$	Estimated response time of device k
$R_{\max}$	Maximum allowerd system response time
R_{high}	Higher threshold, $R_{\text{high}} < R_{\max}$
R_{low}	Lower threshold, $R_{\text{low}} < R_{\text{high}}$

Table 7. Summary of the notation used in this paper

B Queueing Network Models

QN models [16, 42] are a mathematical modelling approach in which a software system is represented as a collection of: (i) *service centers*, which model system resources, and (ii) *customers*, which represent system users or "requests" and move from one service center to another one. The customers' competition to access resources corresponds to queueing into the service centers.

The simplest queueing network model includes a single service center (see Fig. 12) which is composed of a single queue and a single server: the queue models a flow of customers or requests which enter the system, wait in the queue if the system is busy serving other requests, obtain the service, and then depart. Note that, at any time instant only one customer or request is obtaining the service. Single service centers can be described by two parameters: the requests *arrival rate*, usually denoted by λ, and the average *service time* S, i.e., the average time required by the server in order to execute a single request. The maximum service rate is usually indicated with μ and is defined as $\mu = 1/S$. Given the request arrival rate and the requests service time, QN theory allows evaluating the average value of performance metrics by solving simple equations.

In real systems, requests need to access multiple resources in order to be executed; accordingly, in the model they go through more than a single queue. A QN includes several service centers and can be modeled as a directed graph where each node represents the k-th service center, while arcs represent transitions of customers/requests from one service center to the next. The set of nodes and arcs determines the network topology.

Product form QN One of the most important results of queueing network theory is the *BCMP theorem* (by Baskett, Chandy, Muntz, and Palacios-Gomez) which, under various assumptions (see [43] for further details), shows that performance of a software system is independent of network topology and requests routing but depends only on the requests arrival rate and on the requests *demanding time* $D[k]$, i.e., the average overall time required to complete a request at service center k. The average number of time a request is served at the k-th service center is defined as the number of visits $V[k]$, and it holds $D[k] = V[k] \times S[k]$.

In time sharing operating systems, as an example, the average service time is the operating system time slice, while the demanding time is the overall average CPU time required for a request execution. The number of visits is the average number of accesses to the CPU performed by a single request.

Queueing networks satisfying the BCMP theorem assumptions are an important class of

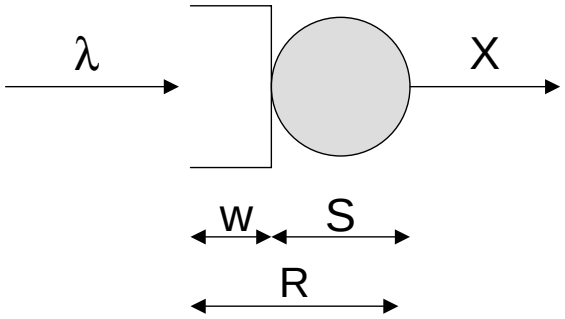


Figure 12. Single Service Center Model. λ denotes the incoming workload, X denotes the requests throughput. W indicates the requests' waiting time, i.e. the average time spent by requests in the queue.

models also known as *separable queueing networks* or *product-form models*[7]. The name "separable" comes from the fact that each service center can be separated from the rest of the network, and its solution can be evaluated in isolation. The solution of the entire network can then be formed by combining these separate results [16, 43]. Such models are the only ones that can be solved efficiently, while the solution time of the equations governing non-product-form queueing network grows exponentially with the size of the network. Hence, in practical situations the time required for the solution of non-product-form networks becomes prohibitive and approximate solutions have to be adopted.

Open and Closed Models Queueing models can be classified as *open* or *closed* models. In open models customers can arrive and depart and the number of customers in the system cannot be determined a-priori (and, possibly, it can be infinite). The single service center system in Fig. 12 is an open model. Vice versa, in closed models the number of customers in the system is a constant, i.e., no new customer can enter the system, and no customer can depart. While open models are characterized by the requests arrival rate λ, a closed model can be described by the average number of users in the system N and by their *think time* Z, i.e., the average time that each customer "spends thinking" between interactions (e.g., reading a Web page before clicking the next link). Customers in closed queue models are represented as delay centers.

Single and Multi-Class Models Finally, queueing models can be classified as *single-class* and *multi-class* models. In single-class models, customers have the same behaviour; vice versa, in multi-class models, customers behave differently and are mapped into multiple-classes. Each class c can be characterized by different values of demanding time $D[c,k]$ at the k-th service center, different arrival rate $\lambda[c]$ in open models, or number of users $N[c]$ and think time $Z[c]$ in closed models. Customers in each class are statistically indistinguishable. Queueing network theory allows determining performance metrics (i.e., response times, utilizations, etc.) on a per-class basis or on an aggregated basis for the whole system.

Solution Techniques After modelling a software system as a queueing network, the model has to be evaluated in order to determine quantitatively the performance metrics.

A first step in the evaluation can be achieved by determining the system bounds; specifically, upper and lower bounds on system throughput and response time can be computed as functions of the system workload intensity (number or arrival rate of customers). Bounds usually require a very little computational effort, especially for the single class case [16, 44].

7. The name "product-form" comes from the fact that the stationary state distribution of the queueing network can be expressed as the product of the distributions of the single queues and avoids the numerical solution of the underlying Markov chain.

More accurate results can be achieved by solving the equations which govern the QN behaviour. Solution techniques can be classified as *analytical methods* (which can be *exact* or *approximate*) and *simulation methods*. Exact analytical methods can determine functional relations between model parameters (i.e., request arrival rate $\lambda[c]$, number of customers $N[c]$ and think time $Z[c]$, and requests demanding times $D[c,k]$) and performance metrics. The analytical solution of open models system is very simple even for multiple class models and yields the average value of the performance metrics. The exact solution of single class closed models is known as the *Mean Value Analysis* (MVA) algorithm and has a linear time complexity with the number of customers and the number of service centers of the models. The MVA algorithm has been extended also to multiple classes, but the time complexity is non-polynomial with the number of customers or with the number of service centers and classes [16]. Hence, large closed models are solved by recurring to approximate solutions, which are mainly iterative methods and can determine approximate results in a reasonable time. Approximate MVA algorithms for multi-class closed models provide results typically within a few percent of the exact analytical solution for throughput and utilization, and within 10% for queue lengths and response time [16].

Analytical solutions can determine the average values of the performance metrics (e.g., average response time, utilization, etc.) or, in some cases, also the percentile distribution of the metric of interest. Determining the percentile distribution of large systems is usually complex even for product-form networks. Indeed, while the mean value of the response time of a request that goes through multiple queues is given by the sum of the average response time obtained locally at the individual queues, the aggregated probability distribution is given by the convolution of the probability distribution of the individual queues. The analytical expression of the percentile distribution becomes complicated for large system (most of the studies provided in the literature are limited to *tandem queues*, i.e., queueing networks including two service centers [45, 42]).

C MIP Problem in GNU MathProg

We describe the MIP model of Fig. 6 using the GNU MathProg language [46], a subset of the AMPL modeling language [47]; the model can be solved using GLPK [48]. Note that in the model below we assume that all devices support the same numer `nL` of ACPI performance states; again, this has been done just for notational convenience, since EASY has been presented in this paper for the general case of heterogeneous devices, and it is also possible to describe the general MIP problem (although less concisely) in AMPL/MathProg.

```
##### Input parameters #########

param N, integer, > 0; # Num. of requests
param nK, integer, > 0;# Num. of devices
param nL, integer, > 0;# Num. of power levels
param Rmax > 0;        # Max response time

set K := 1..nK;        # Devices
set L := 1..nL;        # Performance states

# ENE[k,l] is the ]energy consumption
# rate of device k in performance state l
param ENE{K, L} > 0;

# DD[k,l] is the estimated service demand
# of device k in performance state l
param DD{K, L} > 0;

##### Variables ################
```

```
# X[k,l] = 1 iff device k is
# set at performance level l
var X{K, L}, binary;

var D{K} >= 0;     # service demands
var Dmax >= 0;     # max service demand
var Dtot >= 0;     # sum service demands
var Dave >= 0;     # average service demand
var R >= 0;        # response time
var Rplus >= 0;    # upper bound of R
var Rminus >= 0;   # lower bound of R

##### Optimization problem #####

# Objective function: minimize energy
#  consumption rate
minimize energy_consumption:
    sum{k in K, l in L} X[k,l]*ENE[k,l];

##### Constraints ##############

# Response time must be less tha Rmax
s.t. response_time: R <= Rmax;

# Each device must be put in
# exactly one performance state
s.t. one{k in K}: sum{l in L} X[k,l]=1;

# define response time
s.t. comp_R: R = (Rplus + Rminus) / 2;

# define upper bound on response time
s.t. comp_Rplus: Rplus = Dtot + (N-1)*Dmax;

# define lower bound on response time
s.t. comp_Rminus1: Rminus >= N*Dmax;
s.t. comp_Rminus2: Rminus >= Dtot + (N-1)*Dave;

# define maximum service demand
s.t. comp_Dmax {k in K}: Dmax >= D[k];

# Define total service demand
s.t. comp_Dtot: Dtot = sum{k in K} D[k];

# Define average service demand
# (card(K) is the cardinality of set K)
s.t. comp_Dave: Dave = Dtot / card(K);

# Define service demands D[k]
s.t. comp_D{k in K}:
    D[k] = sum{l in L} X[k,l]*DD[k,l];
```