Evaluating load balancing policies for performance and energy-efficiency

Freek van den Berg *
University of Twente
Enschede, The Netherlands
f.g.b.vandenberg@utwente.nl

Björn F. Postema †
University of Twente
Enschede, The Netherlands
b.f.postema@utwente.nl

Boudewijn R. Haverkort *†
University of Twente
Enschede, The Netherlands
b.r.h.m.haverkort@utwente.nl

Nowadays, more and more increasingly hard computations are performed in challenging fields like weather forecasting, oil and gas exploration, and cryptanalysis. Many of such computations can be implemented using a computer cluster with a large number of servers. Incoming computation requests are then, via a so-called load balancing policy, distributed over the servers to ensure optimal performance. Additionally, being able to switch-off some servers during low period of workload, gives potential to reduced energy consumption. Therefore, load balancing forms, albeit indirectly, a trade-off between performance and energy consumption. In this paper, we introduce a syntax for load-balancing policies to dynamically select a server for each request based on relevant criteria, including the number of jobs queued in servers, power states of servers, and transition delays between power states of servers. To evaluate many policies, we implement two load balancers in: (i) iDSL, a language and tool-chain for evaluating service-oriented systems, and (ii) a simulation framework in AnyLogic. Both implementations are successfully validated by comparison of the results.

1 Introduction

In 2006, Al Gore created a global awareness and willingness to reduce greenhouse gasses by releasing his film "An inconvenient truth" [14]. Roughly one third of these green house gasses are caused due to the generation of electricity[1]. Additionally, approximately 1.1%-1.5% of the worldwide electricity was consumed by data centres in 2010 [11]. In 2012, Neelie Kroes [12], former vice-president of the European Commission responsible for the digital agenda, states that ICT consumes 8% to 10% of all European electricity, which is approximately the total power consumption of whole of the Netherlands. All of the above has led to a stronger focus on green ICT solutions in which saving electricity has becoming increasingly important.

A major energy consumer in ICT are server farms. These server farms consist of many servers that take care of some load. A server environment is set up in a data centre, which provides the infrastructure that enables servers to function. These data centres often consist of many more components that consume energy. Because the energy consumption of these components positively correlates with power consumed by the servers, even small improvements made at server level have a strengthened effect (the so-called "cascade-effect" [6]).

*This research was supported as part of the Dutch national program COMMIT, and carried out as part of the Allegio project under the responsibility of the Embedded Systems Innovation group of TNO, with Philips Medical Systems B.V. as the carrying industrial partner.

†The work in this paper has been supported by the Dutch national STW project Cooperative Networked Systems (CNS), as part of the program "Robust Design of Cyber- Physical Systems" (CPS).

[1] http://www3.epa.gov/climatechange/ghgemissions/sources.html

Submitted to:
QAPL 2016

One of the ways to reduce energy consumption is achieved with the aid of power management features. *Power management* allows to switch between power states of servers to reduce power consumption, while trying to keep the performance intact (e.g., bringing to sleep underutilised servers). There are two key elements that construct a power management *policy*, namely: (i) *strategies* and (ii) *load balancing*. First, power management strategies describe when servers should switch between the power states. Second, the load should be balanced among the servers such that optimal performance is obtained.

This paper proposes a powerful, yet concise, policy language, which covers, among others, strategies that observe the size of the queue to decide to which server jobs are assigned. Furthermore, servers are put to sleep when idling with a simple time-out mechanism, which should be easy to implement in actual servers as literature suggests [4]. The method proposed in this paper allows us to explore a large set of designs by adjusting only three parameters: (i) *queue size threshold*, (ii) *idle time-out*, and (iii) *non-determinism resolution*. In the end, this leads to interesting power-performance trade-offs, i.e., we explore the possibility to exchange reduction of power consumption at the cost of performance.

The policies are implemented as extensions to iDSL and AnyLogic so that policies can be automatically evaluated. This provides insights in the effectiveness of the policies with respect to performance and energy consumption. Also, both implementations are validated by comparison of the results.
In this paper, we answer the following main research question:

How to obtain high-performance, energy-efficient load balancing policies?

The combined answers to the following research questions answer this main research question.

$\mathscr{K}$1 What constitutes a formal model of a load balancer?

- $\mathscr{K}$1a How to model the performance and energy of a load balancer?
- $\mathscr{K}$1b How to model a load balancer policy?

$\mathscr{K}$2 How to automatically compare the performance and energy of many designs?

- $\mathscr{K}$2a How to evaluate many designs to find good policies regarding performance and energy consumption?
- $\mathscr{K}$2b How to validate the evaluated results of the performance and energy?

This paper is further organised as follows. Section 2 provides the system description of a load balancer. Section 3 formalises a load balancer (question $\mathscr{K}$1), including an energy and performance model (question $\mathscr{K}$1a), as well as a load balancer policy model (question $\mathscr{K}$1b). Section 4 then provides two implementations of a load balancer, using iDSL and Anylogic respectively, which are both used to evaluate many different designs (question $\mathscr{K}$2a). The results of these evaluations are then compared in Section 5 to assess their validity (question $\mathscr{K}$2b). Finally, Section 6 concludes the paper.

2 System description of load balancers

In this section, a system description of load balancers is provided. Section 2.1 introduces the stakeholders and their priorities. Section 2.2 presents a data centre performance cluster, which is then simplified by introducing a scope and assumptions. Section 2.3 provides properties that constitute an effective policy.

2.1 Stakeholders and their concerns

In general, load balancing distributes workload among various computing resources, e.g., computers, network links or processing units. Especially server clusters designed for computing have dedicated

equipment for balancing load among its servers. This equipment is then often allocated in a data centre, which is a facility used to house computer systems and associated components.

From the perspective of the data centre infrastructure suppliers we distinguish two major qualities: (i) *customer demands* and (ii) *supplier demands*. According to [2], the system architecture in data centres is mostly driven by five customer demands: availability, scalability, flexibility, security and performance. Insight into these demands is essential for the quality of the data centre. Infrastructure suppliers, however, focus mainly on *energy consumption*, *IT equipment value* and *staff required*. The total energy consumed in a data centre depends on the energy consumed by its switching gear, batteries, power distribution, servers, chillers, coolers, network equipment and monitoring and control devices.

In computing, performance is the most essential demand for customers and energy consumption of these often very expensive server clusters is really high. So especially in the case of computing, a smart load balancer in combination with power management features offer great opportunities to reduce energy consumption, while performance is kept intact.

2.2 Data centre performance cluster

The work in this paper is inspired by the so-called Peregrine cluster at the Center for Information Technology [2] (CIT) in Groningen, the Netherlands, as follows. Assume the CIT decides to actively use power management features for their Peregrine cluster [3] in combination with load balancing. This cluster has a total of 4368 cores with three types of nodes, namely: (i) 162 standard nodes with 2×24 Intel Xeon 2.5 GHz cores; (ii) 6 standard nodes equipped with accelerator cards; and (iii) 7 fast nodes with 4×48 Intel Xeon 2.6 GHz cores. Each standard node consumes approximately 40 W for only the CPU cores[4].

The system of this case study is too complex and knowledge is missing to make a valid statement about performance and energy characteristics. For these reasons, we model the system using the following assumptions. ($\mathscr{A}1$ - $\mathscr{A}10$):

- $\mathscr{A}1$ Incoming requests arrive according to a Poisson process with a negative exponential distribution with rate 1 request per second.
- $\mathscr{A}2$ The system consist of four similar resources (or server nodes).
- $\mathscr{A}3$ A load balancing policy specifies how incoming requests are distributed over these four servers.
- $\mathscr{A}4$ Servers have four power states each; they are either switched on (“stateOn”), asleep (“stateSleep”), moving from “stateOn” to “stateSleep” (“stateSuspend”), or moving from “stateSleep” to “stateOn” (“stateWake”).
- $\mathscr{A}5$ A server can only process tasks in state “stateOn”.
- $\mathscr{A}6$ Servers spend exactly 10 seconds in transition states “stateSuspend” and “stateWakeup” each, when changing states.
- $\mathscr{A}7$ Servers each consume 200 Watt in states “stateOn”, “stateSuspend” and “stateWakeup”, and 14 Watt in state “stateSleep”, based on empirical studies [3, 8]
- $\mathscr{A}8$ Servers have infinite queues that adhere to a non-preemptive FIFO scheduling policy.
- $\mathscr{A}9$ Servers process incoming requests deterministically with rate 1 request per second.
- $\mathscr{A}10$ Evaluating a load balancing policy takes no time.

[2] Center for Information Technology, http://www.rug.nl/society-business/centre-for-information-technology/
[3] Peregrine HPC cluster, https://redmine.hpc.rug.nl/redmine/projects/peregrine
[4] Intel Ark, http://ark.intel.com/

2.3 Effectiveness of a load balancing policy

The effectiveness of a load balancing policy can be seen as a trade-off between performance properties and power properties.

Performance A policy distributes incoming service requests over a number of servers. The way of distributing strongly affects multiple performance metrics, e.g., the queue sizes and utilization of a specific resource are generally high when the load balancer distributes many requests to the same server. In turn, this increases the latencies for requests that are processed by this server. In this paper, only the (average) latency is considered, because it is an interesting and easy to understand metric for the customer.

Energy consumption The way a policy distributes incoming service requests indirectly affects energy consumption, viz., when a policy does not distribute requests to any server for a specified amount of time, the server will go to sleep and use only a fraction of energy. In this paper, we consider the average amount of energy per second (in Watt) the four servers uses *together*.

3 A performance and energy model for load balancers

In the previous section we described load balancers. Here we provide a model to evaluate the performance and energy consumption of load balancing policies. Section 3.1 defines performance and energy consumption by considering incoming requests, power states and transitions, and latencies of requests. Section 3.2 specifies load balancer policies using a grammar and semantics, some typical examples, mechanisms to resolve non-determinism, and the design space.

3.1 Specifying the performance and energy consumption of a load balancer

We define the performance and energy consumption of load balancers in three steps. Section 3.1.1 considers incoming requests and their distribution over resources. Section 3.1.2 shows how power states and transitions depend on incoming requests. Section 3.1.3 specifies latencies of requests.

3.1.1 Incoming requests and their distribution over the resources

Requests arrive with a negative exponentially distributed interarrival time, with rate 1 at the load balancer. Let $I(t)$ indicate that a request arrived at time t. Requests have a unique arrival time. Then $I : 2^{\mathscr{R}}$ is a infinite set with the arrival times of all incoming requests. For illustration, the following I has been generated using a random number generator:

$$I = \{0.87, 0.91, 1.46, 2.03, 3.54, 4.68, 5.42, 5.52, 5.66, 7.26, 9.61, 10.34, 10.93, 11.65, \cdots\}. \quad (1)$$

The incoming requests are inspected by the load balancer, which distributes the requests over selected servers for processing, based on a policy. Let $\mathscr{P}(t) : \mathscr{R}^+ \to \{S_1, S_2, S_3, S_4\}$ be a load balancer policy that distributes the request that arrived at time t to either server S_1, S_2, S_3 or S_4. Let $S_m = \{t \in I | \mathscr{P}(t) = S_m\}$ be the incoming requests of server m. Hence, $\{S_1, S_2, S_3, S_4\}$ is a partition of I, For illustration, assume policy $\mathscr{P}$ distributes the above incoming requests I over the four servers, as follows.

$$\begin{aligned} S_1 &= \{0.91, 3.54, 5.42, 10.34, \cdots\}, & S_2 &= \{9.61, 13.04, 13.52, \cdots\}, \\ S_3 &= \{2.03, 4.68, 5.66, 7.26, \cdots\}, & S_4 &= \{0.87, 1.46, 5.52, 12.01, \cdots\}. \end{aligned} \quad (2)$$

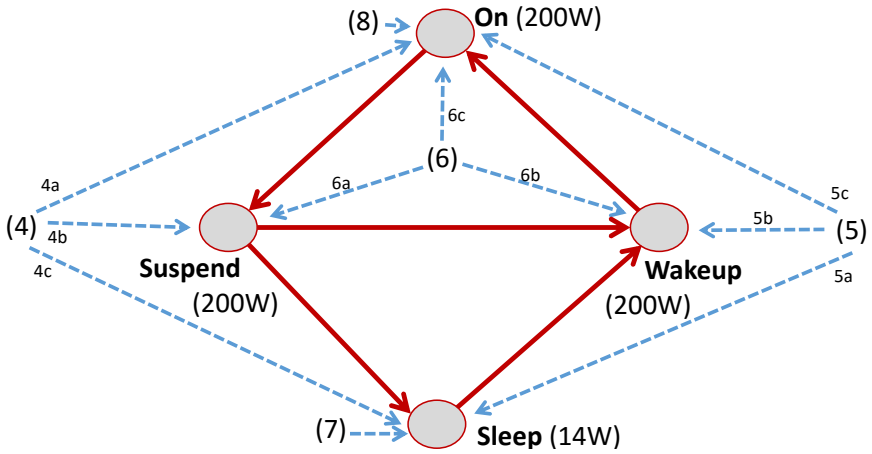


Figure 1: Power states of resources and their transitions

3.1.2 Power state transitions and energy consumption

Servers are in exactly one power state at a time, viz., *on*, suspend (*sd*), sleep (*sl*), or wakeup (*wu*):

$$P_x^{on}(t) \oplus P_x^{sd}(t) \oplus P_x^{wu}(t) \oplus P_x^{sl}(t), \tag{3}$$

where $P_x^s(t)$ indicates that resource x is in state s at time t.

The current power state of a server is implicitly determined by the amount of incoming requests it receives and how quickly it processes them. E.g., when a server receives less incoming requests, it is more likely to go to sleep to save energy.

Next, we present the specification of a load balancer with respect to power state behaviour in equations (4-8). Figure 1 visualizes the four power states as well as the roles the equations play with them, i.e., equation (4-6) are concerned with power three states, and equations (7) and (8) with one power state.

When a server finishes processing its last request and when no new requests arrive in the next TO seconds, the server stays on for TO seconds (4a), suspends for the next 10 seconds, (4b), and ends in sleep mode (4c), as follows.

$$\begin{aligned}(QS_x(t) = 1 \wedge QS_x(t+\varepsilon) = 0 \ \wedge \ S_x \cap [t:t+TO] = \emptyset) \rightarrow \\ (\underbrace{t' \in [t:t+TO] \rightarrow P_x^{on}(t')}_{4b} \ \wedge \ \underbrace{t' \in [t+TO:t+10+TO] \rightarrow P_x^{sd}(t')}_{4b}), \ \wedge \ \underbrace{P_x^{sl}(10+TO+\varepsilon)}_{4c}),\end{aligned} \tag{4}$$

where TO is the time of inactivity needed before a server goes to sleep, $QS_n(t)$ is the queue size plus the request receiving service (either 0 or 1) of server n at time t, and $P_x(t)$ the power state (either on, sleep, suspend, wakeup) of server x at time t.

When a server is in sleep mode (5a) and a request arrives, it wakes up for 10 seconds (5b) and then turns on (5c), as follows.

$$\begin{aligned}(QS_x(t) = 0 \wedge QS_x(t+\varepsilon) = 1 \ \wedge \ P_x^{sl}(t)) \rightarrow \\ (\underbrace{t' \in [t:t+10] P_x^{wu}(t')}_{5b} \ \wedge \ \underbrace{t' \in [t+10:t+10+TO] \rightarrow P_x^{on}(t')}_{5c}).\end{aligned} \tag{5}$$

When a server is suspending (6a) and a request arrives, it will finish suspending and then directly wake up again (6b), as follows.

$$\begin{aligned}(P_x^{on}(t) \wedge P_x^{sd}(t+\varepsilon) \ \wedge \ \exists_{t' \in [t:t+10]} QS_x(t')) \rightarrow \\ (\underbrace{t' \in [t:t+10] \rightarrow P_x^{sd}(t')}_{6a} \ \wedge \ \underbrace{t' \in [t+10:t+20] \rightarrow P_x^{wu}(t')}_{6b})\end{aligned} \tag{6}$$

When a server is in sleep mode (7), it remains there as long as no new requests arrive, as follows.

$$P_x^{sl}(t) \ \wedge \ QS_x(t+t') = 0 \rightarrow P_x^{sl}(t+t') \tag{7}$$

When a server is turned on (8), it remains turned on when incoming requests arrive frequently:

$$P_x^{on}(t) \wedge QS_x(t+t') > 0 \ \wedge \ t' < TO \rightarrow P_x^{on}(t+t'), \tag{8}$$

where TO is the time of inactivity needed before a server goes to sleep. TO is design dependent.

3.1.3 Effectiveness of a load balancer

Section 2.3 addressed performance and energy consumption as the properties to evaluate a load balancer on. The following two equations define, respectively, the performance and energy consumption formally. Performance is defined as the average latency (AL) of all requests conceivable, as follows.

$$AL = \lim_{n \rightarrow \infty} \ \frac{1}{n} \sum_{i=1}^{n} L_i, \tag{9}$$

where L_i is the latency of the i^{th} request.
Average power consumed (AP) to keep all four servers running is defined, as follows.

$$AP = \lim_{t \rightarrow \infty} \sum_{m=1}^{4} \frac{1}{t} \int_0^t \frac{200 \cdot (Pi_m^{on}(s) + Pi_m^{wu}(s) + Pi_m^{sd}(s)) + 14 \cdot Pi_m^{sl}(s)}{4} ds, \tag{10}$$

where $Pi_m^x(t) = 1$ when $P_m^x(t)$, and $Pi_m^x(t) = 0$, otherwise.

3.2 A policy specification for load balancers

A load balancer policy prescribes how a load balancer behaves with respect to distributing incoming requests to servers. Section 3.2.1 defines a policy using a mechanism that orders the servers; Section 3.2.2 presents some example policies. Finally, Section 3.2.3 provides ways to resolve non-determinism when a policy is ambiguous.

3.2.1 A load balancer policy grammar

Each time an incoming requests arrives, a load balancer has to select one of the servers to delegate this request to. We use the following algorithm to make this decision:

- Relevant system state variables are retrieved, e.g., the queue sizes of the servers.
- The preference of each server is determined via an arithmetic expression that includes system state variables.

- The incoming request is delegated to the most desirable server, viz., the server with the highest outcome for the arithmetic expression.

Table 1 shows the grammar of a policy expression $\mathscr{P}$. Policy expressions can be something from the categories *state*, *power*, *time* or *math*, as follows.

$\mathscr{P} = ID \mid numServers \mid queueSize \mid state \mid power \mid time \mid math$
$state = stateOn \mid stateSleep \mid stateSuspend \mid stateWakeup$
$power = powerOn \mid powerSleep \mid powerSuspend \mid powerWakeup$
$time = timeWakeup \mid timeSuspend \mid timeOutTime$
$math = \mathscr{P} * \mathscr{P} \mid \mathscr{P} + \mathscr{P} \mid \mathscr{P} - \mathscr{P} \mid \mathscr{P} / \mathscr{P} \mid \mathscr{P} \mod \mathscr{P} \mid INT \mid random \mid dspace(STRING)$

Table 1: The grammar of load balancer policy expression $\mathscr{P}$

- *State* provides indicator functions to check whether a server is in one of the four states, or not, e.g., when a server is in the *on* state, *stateOn* yields 1 and the others 0.
- *Power* is used to retrieve the power consumptions of each individual state (see assumption $\mathscr{A}7$).
- *Time* includes *timeWakeup* and *timeSuspend*, the time it takes for the server to go back and forth between states *on* and *sleep*, as well as *timeOutTime*, the time of inactivity the server undergoes before going to sleep (see assumption $\mathscr{A}6$).
- *Math* provides five recursive functions that combine policies via arithmetic operations to create arbitrarily complex polices. Furthermore, a constant integer number, a random number $r \in [0:1]$, and a design dependent number could be used.

Furthermore, policy expressions can be an *ID*, a unique number for each server, *numServers*, the total number of servers, and *queueSize*, the number of requests in the queue of each server.

3.2.2 Example policies for load balancers

Let $\mathscr{P}_q$ be a generic policy, where $q \geq 0$ is a variable, to illustrate the functioning of policies in practice:

$$\mathscr{P}_q = -queueSize - q \cdot (1 - stateOn), \tag{11}$$

where q is the server queue size at which an additional server is switched on; q is design dependent.

Then, policy $\mathscr{P}_0$ assigns incoming requests to the server that currently has the shortest queue size:

$$\mathscr{P}_0 = -queueSize \tag{12}$$

Table 2a shows an example evaluation of $\mathscr{P}_0$, where lb_{select} is the choice of the load balancer for a certain server, #n incoming request n, and the numbers in the table the policy evaluations of servers 1-4 for incoming request #n. For the sake of simplicity, we assume that no incoming request finished processing (yet). For request #1-#4, the load balancer arbitrary selects servers, because multiple servers have the highest value for the policy evaluation. In step #5-#8 this pattern repeats. Hence, requests are equally distributed over the servers.

Below, $\mathscr{P}_5$ is also policy that primarily assigns new incoming requests to the server with the shortest queue size. However, $\mathscr{P}_5$ also considers the power state of the servers to save energy. Concretely, it will only switch on a new server if all currently switched-on servers have a queue size of at least 5. Note that this policy might perform less well than $\mathscr{P}_0$, however, at the benefit of reduced energy consumption.

$$\mathscr{P}_5 = queueSize - 5 \cdot (1 - stateOn) \tag{13}$$

	#1	#2	#3	#4	#5	#6	#7
0	0	0	0	0	-1	-1	-1
1	0	0	-1	-1	-1	-2	-2
2	0	0	0	-1	-1	-1	-2
3	0	-1	-1	-1	-1	-1	-1
lb_{select}	3	1	2	0	1	2	3

(a) performance-optimizing policy P_0

	#1	#2	#3	#4	#5	#6	#7	#8
0	-5	-5	-5	-5	-5	-5	-5	-5
1	-5	-1	-2	-3	-4	-5	-5	-5
2	-5	-5	-5	-5	-5	-5	-1	-2
3	-5	-5	-5	-5	-5	-5	-5	-5
lb_{select}	1	1	1	1	1	2	2	2

(b) reasonably energy-efficient policy P_5

Table 2: Example evaluations of policies $\mathscr{P}_0$ and $\mathscr{P}_5$, respectively

	#1	#2	#3	#4
0: random	0.61	0.78	0.05	0.68
1: random	0.46	0.09	0.22	0.79
2: random	0.70	0.12	0.93	0.15
3: random	0.76	0.39	0.51	0.66
lb_{select}	3	0	2	2

(a) random policy $\mathscr{P}' = random$

	#1	#2	#3	#4
0:ID/numservers	0	0	0	0
1:ID/numservers	0.25	0.25	0.25	0.25
2:ID/numservers	0.5	0.5	0.5	0.5
3:ID/numservers	0.75	0.75	0.75	0.75
lb_{select}	3	3	3	3

(b) policy fixed order $\mathscr{P}'' = \frac{ID}{numServers}$

Table 3: Example evaluations of non-determinism resolution mechanisms to two policies

Table 2b show an example evaluation of $\mathscr{P}_5$. When request #1 arrives, the policy evaluation of all servers yields -5 because no servers are turned on. When the load balancer arbitrarily delegates this request to server 1, server 1 switches on and its policy evaluates to -1. Consequently, the load balancer selects server 1 for request #2-5. Then request #6-10 are delegated to server 2 for similar reasons. Note that after 10 incoming requests only two servers have received requests, which is a good property when energy consumption is of concern.

3.2.3 Resolving non-determinism

In this section, we provide a solution that prevents an arbitrary selection of a server by the load balancer when multiple servers have the highest value for their policy expression.

Table 2b shows an example evaluation for $\mathscr{P}_5$. For each incoming request #1-#8, the load balancer selects the server with the highest evaluated value, e.g., for request #2 server 1 gets selected because its policy expression evaluates to -1, which is higher than the -5 of the other three servers. However, there are cases in which the expression of multiple servers has the highest evaluation, e.g., for request #1 all servers evaluate to 1, which makes selecting server 1 an arbitrary decision. In these cases, the load balancer performs a so-called non-deterministic decision.

Non-determinism can be resolved by adding fractions $f \in [0:1)$ to policy outcomes, as follows. Let $\mathscr{P}$ be a policy that only returns whole numbers $\mathscr{N}$, then policies

$$\mathscr{P}'_q = \mathscr{P}_q + random \qquad \mathscr{P}''_q = \mathscr{P}_q + \frac{ID}{numServers} \tag{14}$$

yield unique real numbers $\mathscr{R}$ for each server, eliminating non-determinism. The policy outcomes are only unique, if we assume that randomly drawn numbers are unique and if each server has a unique ID.

We illustrate how these two resolution mechanisms work by providing two example evaluations for them, respectively. For the sake of simplicity, we use policies $\mathscr{P}'. = random$ and $\mathscr{P}'' = \frac{ID}{numServers}$.

Table 3a shows how $\mathscr{P}'$ functions. For each incoming request, four random numbers are drawn and the load balancer delegates the request to the server with the highest value, e.g., request #1 is delegated to server 3 because $max(0.61, 0.46, 0.70, 0.76) = 0.76$.

Table 3b shows how $\mathscr{P}''$ functions. For each incoming request, the policy evaluates to a unique number per server, which is divided by 4, the number of servers, to return number in range $[0, 1)$. The load balancer delegates all requests to server 3 that has the highest ID, namely 3.

3.2.4 Design space

We define a design space to compare many policies. Each design then represents a load balancer with a different policy. The design space is defined as the Cartesian product over the following three dimensions and their ranges.

- $q \in \{1, 2, 3, 5, 7, 10, 15, 20, 30, 40, 50, 75, 100\}$ (cf. equation 11).
- timeout time: $TO \in \{1, 2, 3, 4, 5, 7.5, 10, 15, 30\}$ (cf. equation 8).
- non-determinism resolution: $nd \in \{random, fixed\_order\}$ (cf. equation 14).

E.g., design $d = (5, 10, random)$ is a design in which: (i) a new server is turned on when the queue sizes of all currently running servers is greater than 5, (ii) servers shut down after 10 seconds of inactivity, and (iii) non-determinism is resolved via a random selection.

4 Two implementations

In this section we implement the performance and energy model in two different ways to enable the automatic evaluation of many load balancing policies and validate the results. Section 4.1 provides an iDSL implementation, whereas Section 4.2 provides an AnyLogic implementation.

4.1 iDSL implementations

iDSl [19, 21, 20] is a formal language and solution chain to evaluate the performance of service-oriented systems. iDSL has been developed made using Eclipse for DSLs[5] and is, therefore, an Eclipse plug-in with an extensive IDE. iDSL supports both simulation and model checking, via a transformation to Modest [9], as means to evaluate large numbers of complex designs. Finally, iDSL presents its predictions intuitively via visualizations and understandable (aggregated) metrics.

We extend iDSL to support load balancers, as follows. In Section 4.1.1, we extend the iDSL language with a load balancer construct and define its semantics via a transformation to Modest. In Section 4.1.2, we define an iDSL instance with a load balancer.

4.1.1 Extending iDSL to support load balancers

We extend iDSL to support load balancers. A language construct named *lbalt* is generated first (see Figure 3, Section Process). *lbalt* contains a *policy* (as defined in Section 3.2.1), a *configuration* with power consumptions per state and transition times (of Section 3.1.2), and multiple processes that are

[5] http://www.eclipse.org/downloads/packages/eclipse-java-and-dsl-tools/junosr1-rc2

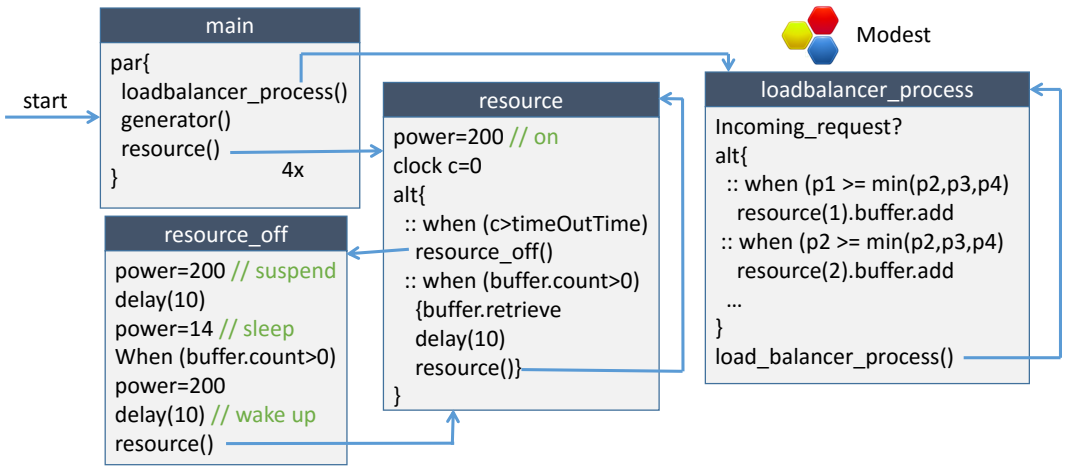


Figure 2: The Modest processes for the load balancer

```
Section Process
  ProcessModel p
    lbalt
      policy { queueSize + dspace("q")*(1-stateOn) + dspace("notdet_res") }
      configuration { powerOn 200 powerShutdown 200 powerSleep 14
                      powerStartup 200 timeStartUp 10
                      timeShutdown 10 timeOutTime dspace("to") }
      { atom p1 load 1, atom p2 load 1, atom p3 load 1, atom p4 load 1}

Section Resource
  ResourceModel r
    decomp { atom r1 rate 1, atom r2 rate 1, atom r3 rate 1, atom r4 rate 1 }

Section System
  Service serv
    Process p
    Resource r
    Mapping assign            { (p1, r1) (p2, r2) (p3, r3) (p4, r4) }
            scheduling policy { (r1, FIFO) (r2, FIFO) (r3, FIFO) (r4, FIFO) }

Section Scenario
  Scenario sc
    ServiceRequest serv with distribution exp "1"

Section Measure
  Measure
    ServiceResponse times using 1 runs of 1500 ServiceRequests

Section Study
  Scenario sc
    DesignSpace
      ("q" {"1" "2" "3" "5" "7" "10" "15" "20" "30" "40" "50" "75" "100"} )
      ("to" {"1" "2" "3" "4" "5" "7.5" "10" "15" "30"} )
      ("notdet_res" {"random" "ID/LBNumServers"} )
```

Figure 3: A full iDSL instance for a load balancer with four servers

mapped to resources to distribute incoming request to. Also, iDSL has been extended to support energy-efficient resources with four power states.

Under the hood, the load balancer and energy-efficient resources are transformed to multiple Modest [9] processes, as follows. Figure 2 shows initial process *main* that initializes a *load balancer* process, a *generator* for incoming requests, and four energy-efficient *resources*, in parallel. To load balancer becomes active when an *incoming request* enter the system, viz., the load balancer evaluates the policy for each resource and adds the incoming request to the buffer of the preferred resource. At the same time, resources wait for a request to arrive in the buffer, which they then process. Alternatively, the resource times out: it goes to *suspend* mode (in process *resource_off*) first, then to *sleep* mode, and finally waits for a request to arrive in its buffer. When this happens, it turns *on* again (by calling process *resource*). Note that the power consumption has been modelled using a reward named power.

4.1.2 IDSL instance of the load balancer and experiments

Figure 3 shows a full iDSL instance of a load balancer with four servers. The iDSL consists of the following six sections. Section *Process* defines a load balancer (as explained in Section 4.1.1) that consists of a policy (based on equation (11), a configuration (based on assumptions $\mathscr{A}6$ and $\mathscr{A}7$), and four processes. In Section *Resource* the four servers are defined. Section *System* then maps the four processes to the four servers, respectively, via a FIFO scheduling policy. In Section *Scenario* the exponential rate of the incoming requests is set to 1. Section *Measure* indicates that simulation runs of 1500 incoming requests each are used. Finally, Section *Study* defines the design space, the ternary Cartesian product of dimensions q, to, and $nondet\_res$. Note that the variables are used in the the policy and configuration of the load balancer via the $dspace$ construct, which means these vary per design.

4.2 AnyLogic implementation

In [17], a simulation framework proposed that allows for the analysis of power and performance trade-offs for data centres that save energy via power management. A combination of high-level models is formed to estimate data centre power consumption and performance. These high-level cooperating simulation models are concerned with (i) IT equipment, (ii) the cascade effect, (iii) the system workload, and (iv) power management. The framework is developed in the AnyLogic [1] multimethod simulation software, which allows the use of a combination of discrete-event and agent-based models. The framework offers an intuitive dashboard to actively control and obtain insight during each simulation run, as illustrated in Figure 4. Besides insight into transient behaviour, as can be seen in this figure for the (a) *power-state utilisation*, (b) *response times* and (c) *power consumption*, also averages are computed and depicted in tables to give an indication of the steady-state behaviour.

In Section 4.2.1, the configuration of the data centre in the simulation framework is elaborated. An extension for policies of load balancing and power management for this framework is introduced in Section 4.2.2.

4.2.1 Configuring the simulation framework

The framework is configured according to the system description (of Section 2). The basic load balancer and its environment are implemented with one agent for the load balancer and one agent for each server. The load balancer distributes the workload by injecting jobs to the servers. These jobs are injected in simple queues inside the server agents.

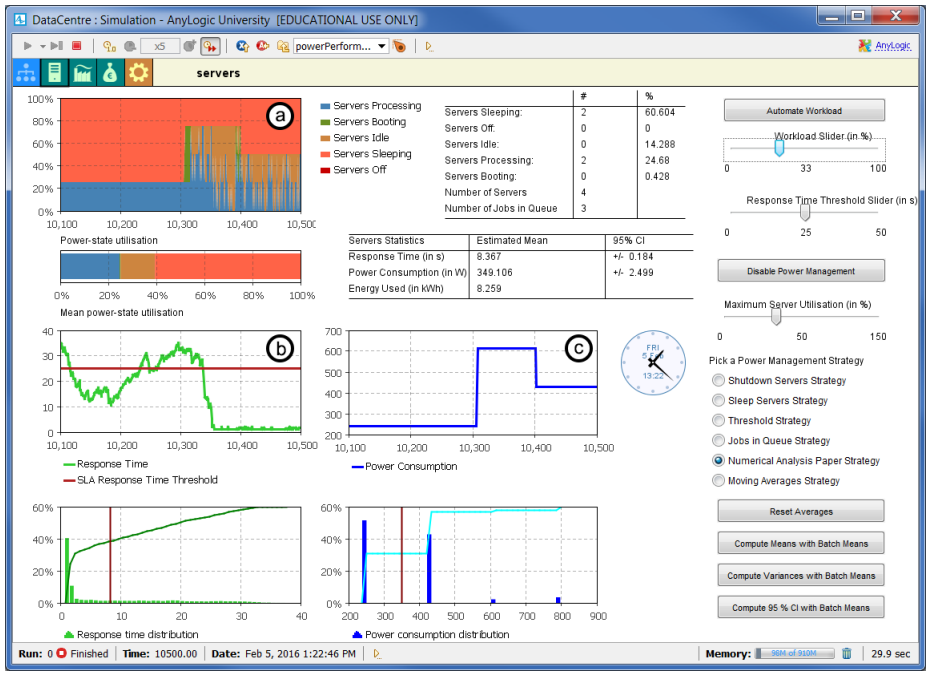


Figure 4: The AnyLogic dashboard

Table 4 shows all the parameters used. Jobs arrive in the load balancer according to a Poisson process with rate (λ job/s). The service rate (μ job/s) of each server is deterministic, i.e., the server finishes jobs in a fixed amount of time. The model is extended to support four power states, cf. (3). The awareness of power states in the models allows to compute power consumption (P) by rewarding each power state with a power consumption. Note that processing is the only power state in which jobs are served.

P_{pc}	200 W	P_{as}	14 W
P_{sl}	200 W	P_{sl}	200 W

(a) power management parameters

λ	exp(1.0)	μ	det(1.0)
α_{sl}	det(10)	α_{wk}	det(10)

(b) performance parameters

Table 4: Data centre configuration parameters

The mean power consumption ($E[P]$) and mean response times ($E[R]$) are computed using the batch means method. The batch means method requires the length of the simulation (t_{sim}) to be very long, which is usually around 100,000 virtual seconds, and the system should be stable after some warm-up (wup) period, which is usually around 500 virtual seconds.

4.2.2 Policy implementation

Recall that the load balancer injects jobs in the queues of agents of the servers. Therefore, the load balancer needs a policy to determine where each job should go. The policies, introduced in [17], have two simple options: (i) random and (ii) shortest queue next. So, the load balancer required an extension to support policies (cf. Section 4.2.2).

In order to implement these policies, information is required about the size of the queue of each server and about the current power state. In order to select a server that support these policies an expression should be defined to reward each server. The extension consist of a module class that has access to all the

relevant information, such that it can rate the servers based on the three parameters (cf. Section 3.2.4). Additionally, a parameter variation experiment of the framework is implemented that allows for parallel computation of the averages of many designs.

5 Experimental results

We show what results of the evaluated designs tell us about policies (in Section 5.1) and their validity (in Section 5.2).

5.1 Lessons learned

Figure 5 shows the Anylogic results for all designs. It shows the effect of adjusting the parameters in the policy for various values of queue threshold $q \in \{1,2,3,5,7,10,15,20,30,40\}$ and the time-out $TO \in \{1,2,3,4,5,7.5,10,15,30\}$. In Figure 5a equal values for q are marked with the same colour. It shows that high values of q lead to low energy consumption and low values of q to high performance. Figure 5b has similar colours TO. It shows that the efficiency frontier depends on the time-out value TO.

5.2 Validity of the outcomes

We assess the validity of the iSDL and AnyLogic approaches by comparing their performance and energy consumption outcomes for many different designs. The following distance measure, which returns the ratio differences, is used to compare outcomes:

$$\delta(v_1, v_2) = \max\left(\frac{v_1}{v_2}, \frac{v_2}{v_1}\right) - 1 \tag{15}$$

The measure is partly like a metric, viz., $\delta(v,v) = 0$, $\delta(v_1,v_2) = \delta(v_2,v_1)$, and $\delta(av_1,av_2) = \delta(v_1,v_2)$. However, the triangular property $\delta(v_1,v_2) + \delta(v_2,v_3) \geq \delta(v_1,v_3)$ does not hold.

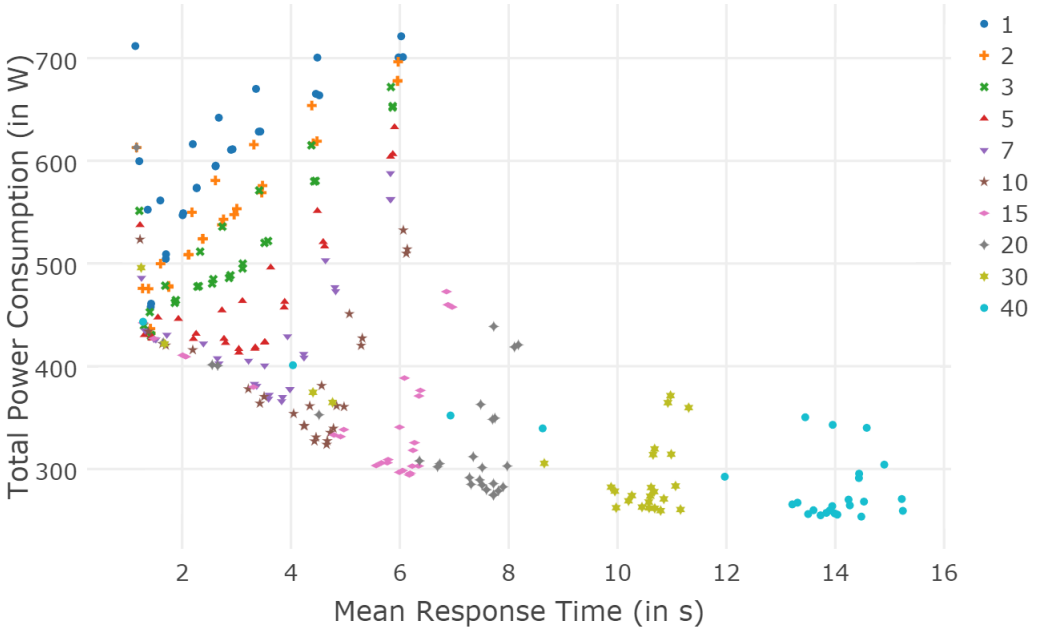
Figure 6 shows the experimental outcomes of iDSL (on the y-axis) and AnyLogic (on the x-axis) for resolving non-determinism with the random (in Figure 6a) or the fixed order (in Figure 6b) way, respectively. Note that the distance δ is visualised around the diagonal for values 0, 0.1, 0.2 and 0.3. Generally, the results of both implementations match, because most designs are located near the diagonal.

6 Conclusions

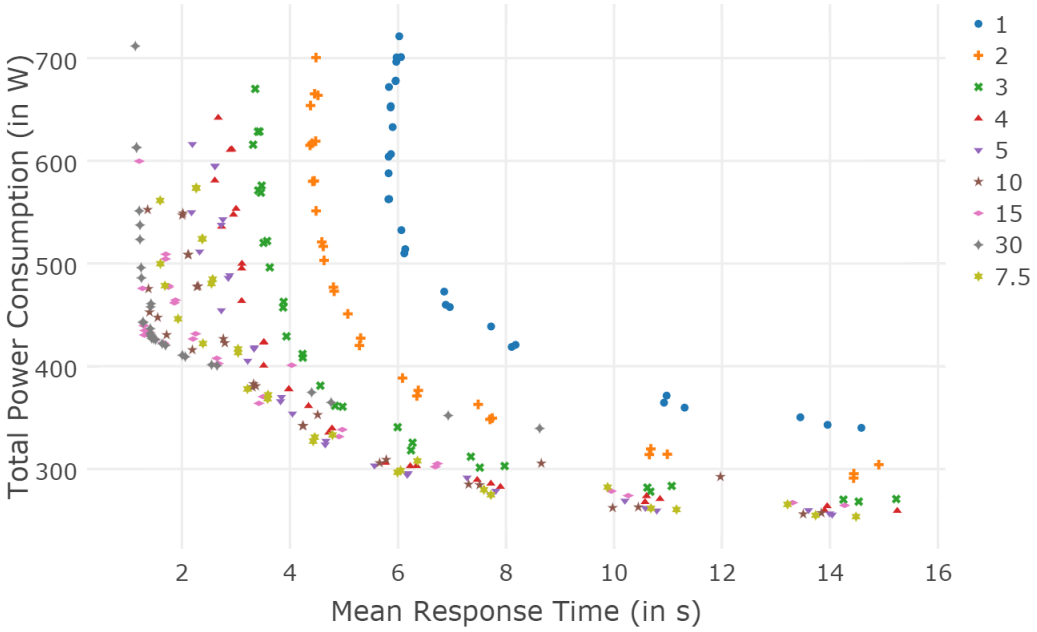
In this paper we have constructed a model to evaluate the performance and energy consumption of load balancers. In this model, we define a powerful policy language that decide to which server jobs are assigned by observing the system variables, e.g., queue sizes of servers.

To evaluate the performance and energy trade-off of many policies, we have implemented two load balancers with exactly the same specifications in iDSL [19, 21, 20] and in AnyLogic [1]. Alternatives for iDSL, which offers a high-level language, are PRISM [10], Modest [9] and UPPAAL [13]. Cloudsim [5] is an alternative for AnyLogic.

Evaluation of many policies shows that parameter q, the queue threshold for switching servers on, is useful to resolve the performance and power consumption trade-off, viz., low q values leads to good performance, while higher q values reduce energy consumption. Parameter TO, the idling time of servers

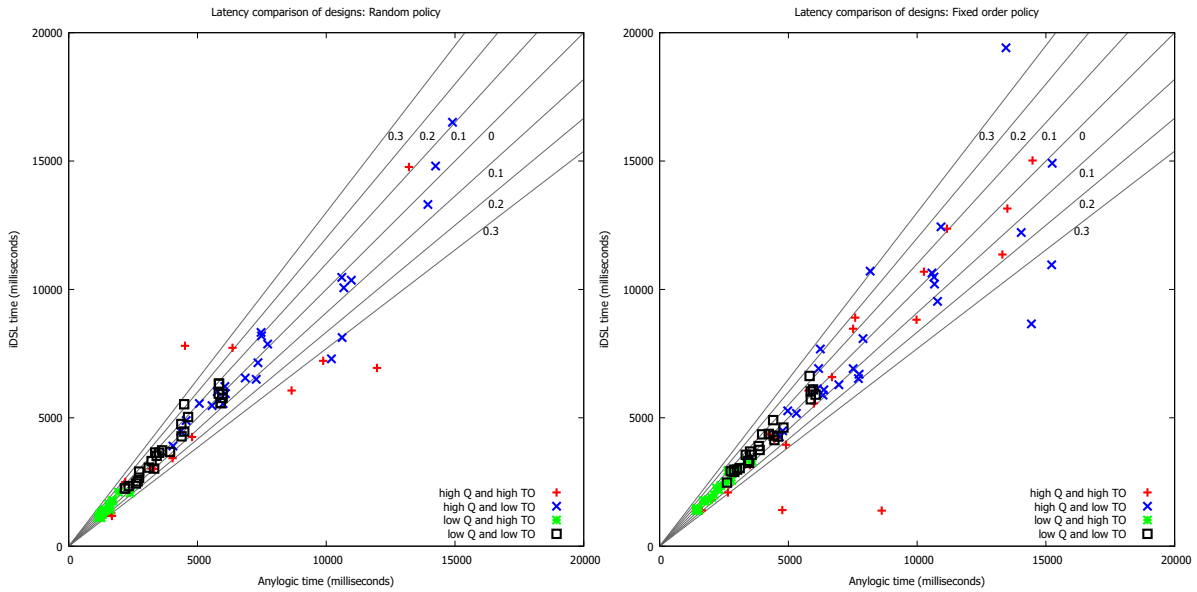


(a) The value q determines the amount of power traded for performance.



(b) Time-out to determines the position of the frontier.

Figure 5: Average latency and power consumption outcomes for many designs



(a) Random non-determinism elimination

(b) Fixed order non-determinism elimination

Figure 6: Comparison of the iDSL (on the y-axis) on AnyLogic (on the x-axis) results for many designs

before sleeping, determines the position of the so-called Pareto optimal frontier. A higher TO value improves both performance and power consumption.

For validation, the evaluated performance and energy consumptions results of both implementations have been compared. For half of all the designs, both the average latencies and power consumption of iDSL and AnyLogic differed less than 6%. For 80% of the designs, this is 13% and 11%, respectively.

Related work The work of [18] simulate models that consider virtual machines and in particular the power-performance trade-off. Similarly, [15] considers virtual machines and a power-performance trade-off with testbed to apply their models for monitoring and control. In [7], power management is discussed with a strong focus on server allocation. Furthermore, [16] performed on cluster-based systems with a load balancer taking power and performance into account. Finally, [5] offers power and performance analysis for data centres.

Our work distinguish itself in the following three ways: First, we have constructed a load balancer policy with a powerful yet concise language which is used to access system variables, such as queue sizes and power states of servers. Second, we have implemented this policy in two different development environments, iDSL and AnyLogic. Both implementations were validated by comparison of evaluated results. Third, evaluation of many designs provides insight in the meaning of the policies while only using two parameters: The queue size threshold the affecting the performance power trade-off, the server idle time affecting the level of Pareto optimality.

References

[1] AnyLogic. AnyLogic: Multimethod Simulation Software, 2000.

[2] M. Arregoces and M. Portolani. *Data Center Fundamentals*. Cisco Press, Indianapolis, 2003.

[3] L. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, dec 2007.

[4] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):299–316, June 2000.

[5] R. Calheiros, R. Ranjan, A. Beloglazov, C. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[6] Emerson Network Power. Energy Logic: Reducing Data Center Energy Consumption by Creating Savings that Cascade Across Systems. *White Paper of Emerson Electric Co*, 2009.

[7] A. Gandhi. *Dynamic Server Provisioning for Data Center Power Management*. Phd thesis, Carnegie Mellon University, 2013.

[8] A. Gandhi, M. Harchol-Balter, and M. Kozuch. Are sleep states effective in data centers? In *Proc. of Int. Green Computing Conference*, pages 1–10. IEEE, jun 2012.

[9] A. Hartmanns and H. Hermanns. A modest approach to checking probabilistic timed automata. In *QEST*, pages 187–196. IEEE, 2009.

[10] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems.

[11] J. Koomey. Growth in data center electricity use 2005 to 2010. *Oakland, CA: Analytics Press. August*, 1, 2011.

[12] N. Kroes. Using ICT to build Sustainable Cities, 2012.

[13] K. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *STTTT*, 1(1):134–152, 1997.

[14] J. Nolan. an inconvenient truth increases knowledge, concern, and willingness to reduce greenhouse gases. *Environment and Behavior*, 42(5):643–658, 2010.

[15] V. Petrucci, E. Carrera, O. Loques, J. Leite, and D. Mossé. Optimized Management of Power and Performance for Virtualized Heterogeneous Server Clusters. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 23–32. IEEE, may 2011.

[16] E Pinheiro, R Bianchini, E. Carrera, and T Heath. Load balancing and unbalancing for power and performance in cluster-based systems. *Work. on compilers and operating systems for low power*, 180, 2001.

[17] B.F. Postema and B.R. Haverkort. An AnyLogic Simulation Model for Power and Performance Analysis of Data Centres. In Marta Beltrán, William Knottenbelt, and Jeremy Bradley, editors, *Computer Performance Engineering*, volume 9272 of *Lecture Notes in Computer Science*, pages 258–272, Madrid, Spain, 2015. Springer International Publishing Switzerland.

[18] H. Van, F. Tran, and J. Menaud. Performance and Power Management for Cloud Infrastructures. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 329–336. IEEE, jul 2010.

[19] F. van den Berg, J. Hooman, A. Hartmanns, B.R. Haverkort, and A. Remke. Computing response time distributions using iterative probabilistic model checking. In *EPEW*, volume 9272 of *Lecture Notes in Computer Science*, pages 208–224. Springer, 2015.

[20] F. van den Berg, A. Remke, and B.R. Haverkort. A domain specific language for performance evaluation of medical imaging systems. In *MCPS 2014*, volume 36 of *OASICS*, pages 80–93. Schloss Dagstuhl, April 2014.

[21] F. van den Berg, A. Remke, and B.R. Haverkort. iDSL: Automated performance prediction and analysis of medical imaging systems. In *EPEW*, volume 9272 of *LNCS*, pages 227–242. Springer, 2015.