Empirical Evaluation of Power Saving Policies for Data Centers

Michele Mazzucco
University of Tartu
Estonia

Isi Mitrani
Newcastle University
United Kingdom

Abstract

It has been suggested that the conflicting objectives of high performance and low power consumption in a service center can be met by powering a block of servers on and off, in response to changing demand conditions. To test that proposition, a dynamic operating policy is evaluated in a real-life setting, using the Amazon EC2 cloud platform. The application running on the cluster is a replica of the English edition of Wikipedia, with different streams of requests generated by reading traces from a file and by means of random numbers with a given mean and squared coefficient of variation. The system costs achieved by an 'optimized' version of the policy are compared to those of a simple heuristic and also to a baseline policy consisting of keeping all servers powered on all the time and one where servers are re-allocated periodically but reserves are not employed.

1 Introduction

Large scale service centers containing hundreds or thousands of computers are no longer a novelty. Their development and widespread use has raised considerable interest in the problem of reducing the amount of power they consume, while at the same time maintaining a satisfactory level of performance (*e.g.*, response time). While different techniques aiming at saving energy exist, with currently available technology the only realistic way to reduce *significantly* the power consumption of a server farm is to power down blocks of servers whenever that can be justified by the demand conditions, hence increasing the system utilization; power capping decreases the power consumption by slowing down the servers, but it does not increase the system utilization, while recent developments in processor and memory technology limit the potential energy savings resulting from DVFS [8].

While there have been proposals for operating policies that work in exactly the way described above, those proposals have been evaluated by means of numerical models and/or simulations only; however, they have not been put to the test in a real-life service center. That is the purpose of the present study. The particular policy we examine is one where a subset of the available servers are left powered up all the time, while the rest are designated as 'reserves' and are powered up and down dynamically, depending on the number of jobs in the system. There are heuristic rules for deciding how big should be the reserve block, and when it should be activated and deactivated. The objective function that the policy attempts to minimize is a linear combination of the cost of the powered-up servers and the cost of holding jobs in the system. The policy was implemented on a cluster of computers using the Amazon EC2 cloud platform and running a replica of the English edition of Wikipedia. The system was subjected to a traffic of requests whose characteristics were varied. The behavior of the heuristic dynamic policy was compared to the 'default' policy of keeping all servers powered on all the time, to a policy that changes the number of running servers according to the predicted user demand but never uses the reserve block, and also to a policy where a greater effort had gone into optimizing the parameters.

A number of server allocation policies taking into account costs and benefits were proposed by Chase et al [4] and Mazzucco and Dyachuk [9]. The possibility of powering down a portion of the network infrastructure was studied by Bolla et al [3], while Urgaonkar et al [15, 18] proposed policies for provisioning multi-tiered applications. All the above approaches, however, are dynamic only in the sense of reacting to changes in (observed or predicted) user demand; they do not react to changes in the system state (*e.g.*, queue size).

The 'reserve block' policy which is the subject of this paper was proposed and analysed by means of a queueing model in Mitrani [11]. A simpler model of that policy was subsequently examined by Schwarz et al [12]. Other, more distantly related studies were carried out by Artalejo et al, [1], Ghandi et al [5] and Slegers et al [13]. In [1], servers are powered up and down one at a time, and at most one server can be in the process of being powered up. This last restriction was relaxed in [5], via an approximate analysis, while [13] considered the problem of controlling power consumption in the presence of on/off arrival streams by means of Monte Carlo simulations.

Peripherally related to these dynamic policies are approaches based on control theory or virtual machine placement (*e.g.*, see Bobroff et al [2], Verma et al [17] and Kumar et al [7]). Such algorithms are very difficult to optimize in realistic scenarios and may require user intervention (*e.g.*, in [7]).

The rest of this manuscript is structured as follows. The following section introduces the model and the heuristic policy. Section 3 describes the setup employed to test our proposal. Section 4 discusses a number of experiments, while Section 5 concludes the paper.

2 The policy

Suppose that the total number of available servers is N. A subset of them, of size n, is designated as 'the main block'; those servers are left powered on all the time, regardless of whether they are busy or idle. The remaining $(N-n)$ servers form the 'reserve block'. The latter is powered on and off dynamically. More precisely, there is an upper threshold U, such that when the number of jobs (requests) in the system increases from U to $(U+1)$, all the servers in the reserve block are powered on. Similarly, there is a lower threshold D, such that when the number of jobs in the system decreases from D to $(D-1)$, all the servers in the reserve block are powered off. Of course, in practice those operations are not instantaneous but take

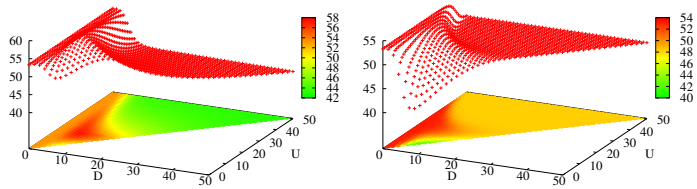
some time during which the reserve servers consume power without being able to serve jobs.
The system performance is measured by the steady state average number of jobs inside the system (*i.e.*, waiting or being served), or, equivalently, by the average response time. Hence, the policy parameters n, U and D should be chosen so as to minimize, as far as possible, a cost function of the form

$$C = c_1 L + c_2 S \,, \tag{1}$$

where L is the average number of jobs in the system and S is the average number of powered on servers; c_1 is the cost of holding one job in the system per second and c_2 is the running cost for one server per second (please note that determining the value of c_1 and c_2 is outside the scope of this paper). Different cost functions are also possible, *e.g.*, one might want to acknowledge the fact that servers executing jobs usually require more energy than idle servers [9].
The queueing model in [11] assumed that if a reserve server was serving a job when the policy decided to power it off, that job would be interrupted and returned to the queue, to be resumed on another available server. In real life, the job would be allowed to complete before the server was powered off. The model also made Markovian assumptions concerning the arrival, service and power-on processes (power-off was assumed to be instantaneous). Based on those assumptions and the resulting analysis, certain heuristics were proposed for choosing the size of the main block, n, and the two thresholds, U and D. In reality the assumptions are unlikely to be satisfied, and the parameters (*e.g.*, the arrival rate) are not known with absolute accuracy, but the heuristics can still be used. The particular values that are employed in our experiments are

$$n = \left\lfloor \rho + \frac{1}{2}\left[1 + \sqrt{1 + 4\rho\frac{c_1}{c_2}}\right]\right\rfloor \;\; ; \;\; U = N \;\; ; \;\; D = n - 1 \,, \tag{2}$$

where $\rho = \lambda b$ is the offered load; λ is the job arrival rate, b is the average service time, and N is the number of available servers. The operation $\lfloor \cdot \rfloor$ is a truncation, while the second term of the equation used to determine n represents the 'safety capacity' used to deal with stochastic variability. While a number of heuristics aiming at deciding the amount of variability edge exist (*e.g.*, see [9]), our strategy relies on approximating the factor L in Equation (1) by the $M/M/1$ expression, see [11].
The rationale behind this policy is to allocate enough servers to cope with the average load, and to make a reasonable choice for what the value of the two thresholds is concerned: the reserves are powered up when the number of jobs in the system exceeds N, while they are powered down when some of the non-reserves become idle. As shown in Figure 1, for a certain value of n, a region where the cost is low for a number of combinations of D and U always exists.
N.B. Even though in this study we optimize all the parameters, the above observation implies that this policy is particularly suitable for scenarios where the main and reserve blocks are fixed (*e.g.*, container based data centers) and one should decide when to power on/off the reserve block.
In addition to the 'Heuristic' policy above, an 'Optimal' policy was implemented, whereby the parameters were obtained by a simulated annealing algorithm searching through the possible values of n, U and D, evaluating the model and selecting the best. Of course, that policy is only optimal if the Markovian assumptions are satisfied. Otherwise, as we shall see, even the default policy can be better.
In order to apply either the 'Heuristic' or the 'Optimal' policy, the parameters λ and b need to be estimated. To that end, the system is monitored and traffic statistics are collected. The observation period is divided into intervals of fixed size: the arrival rate obtained during one such interval (or a transformation of that value, *e.g.*, if a prediction algorithm is employed) is used to determine the policy during the next interval. The average service time (or, rather, its reciprocal, the average service rate $\mu = 1/b$ jobs/sec), instead, is assumed to be fixed and known. Concerning the experiments which will be discussed later, the average service time was determined a priori by estimating the average response time of an over-provisioned system. The obtained value was validated by estimating the average number of jobs inside the system and then employing Little's law to determine the average response time. Since no queueing delay occurs when the system is over-provisioned, the response time is equal to the service time. With regard to this point, it is worth noting that Amazon EC2 instances have two IP addresses. The network delay is negligible *only* if the internal IP address is employed, as public IP address network traffic goes through many routers/hops.
Finally, while the model assumes a system composed of a single tier only, this assumption is lifted during experiments.
N.B. The policy described in Equation (2) assumes that the average offered load is less than n. As we shall see in Section 4.3, this has important implications if the load can not be estimated with absolute accuracy.



(a) $n = 18$ (b) $n = 29$

Figure 1: Cost for different values of U and D. $\lambda = 70$. The configuration achieving the lowest possible cost, 40.687 \$/sec, is $n = 20$, $D = 22$, $U = 45$.

3 Experimental Setup

Rather than employing benchmark applications such as those proposed by SPEC or TPC, we decided to test our policies on a replica of the English edition of Wikipedia, which constitutes about half of the requests received by Wikipedia [14] deployed on the Amazon Elastic EC2 cloud compute platform.
All nodes are deployed on `c1.medium` instances running Ubuntu Linux 11.04 32 bits with kernel 2.6.38. Our setup consists of one virtual machine running HAProxy 1.4.20 to distribute incoming requests to a variable number of Apache 2.2.17 servers running the MediaWiki application. On the same node we also run a Python daemon that connects to HAProxy via UNIX socket; it collects statistics and updates the size of the reserves block as well as its state in order to reflect changes in user demand. Since we are employing Amazon EC2 instances and not bare-metal servers, reserves are always on, but they are enabled/disabled at runtime by changing the configuration of HAProxy. Persistent storage is provided by one MySQL 5.1.54, while another machine runs Memcached 1.4.5, a memory caching system used to speed up the application.
Main optimizations We have set the socket timeout to 20 seconds on both the load balancer and client with the aim of preventing situations where a long request causes a timeout while it is being executed. Also, since we are serving dynamic content only (static content is not an issue – in one test we were able to serve about 0.5 Gbit/sec of static content with just four servers [10]), the workload is rather CPU intensive (PHP caching is disabled); hence, we have

set the maximum number of concurrent connections to each web servers to two. This prevents overloading the web servers, as we have found that two connections suffice to keep the average CPU utilization at about 70%. One might be able to increase the throughput slightly by allowing more concurrent jobs to be served by each Apache server at the expense of very unpredictable response times. Hence, we treat each virtual machine as two servers. Since servers become available and are powered off as a whole, in case an odd value of n is chosen, Equation (1) is evaluated for both $(n-1)$ and $(n+1)$ (with the corresponding 'best' thresholds) and the solution providing the lowest cost is selected.

Dataset The database was initialized with the MediaWiki page dumps of January 15, 2011, consisting of 166,977 articles. These correspond to about 2.8 GB of filesystem space. The operational dataset, however, is composed of about 40,000 articles (excluding the redirects), as we are requesting only a portion of randomly selected articles in order to ensure that most of the requests can be served from the cache (about 75% of the queries are cached by MySQL while the cache hit ratio of memcached is 97%). Using the whole dump would require us to use a distributed deployment for memcached, without introducing any substantial difference.

The reader should keep in mind that adding an additional layer providing a caching abstraction complicates the setup considerably, as jobs might be served entirely from the cache (at least one database access is required even in case of cache hit), entirely from the database, or a mix of the two previous options. The average size of each request is about 59 KB, the average service time is 230 ms, while the squared coefficient of variation of service times is about 0.5 (disabling PHP caching increases the jobs size while at the same time it considerably decreases the service time variability).

Load Generator The load was generated by a customized version of WikiBench [16] running on OpenJDK 1.7; the main difference compared to the original version is that our workload generator uses an open system model (*i.e.*, new job arrivals arrive independently of job completions), which is implemented by means of non blocking I/O. URLs were selected at random by means of an uniform distribution, while interarrival intervals were generated using exponential or lognormal distributions with a given mean and squared coefficient of variation. In order to reduce the network delays all the machines, including the load generator, were deployed on the same availability zone.

4 Experimental Evaluation

In this section we present the results of a number of experiments that were carried out in order to evaluate the performance of our power-saving policies.

As mentioned in Section 2, determining suitable values for c_1 and c_2 is outside the scope of the paper. However, it is worth noting that they affect the choice of n (and consequently the value of of D, see Equation (2)). In particular, the size of the always-on block increases as the ratio c_1/c_2 increases.

When not specified otherwise, the following parameters are employed: c_1 is 1.2 \$/sec, c_2 is 1 \$/sec, the average service rate is 4.35 jobs/sec, 34 CPUs (*i.e.*, the saturation point occurs at $34 \times 4.35 = 147.9$ jobs/sec), and the average power-up delay is 60 seconds. Before embarking on the real-life trials, we evaluated the model in [11] numerically, under Markovian assumptions, so as to examine the extent to which the achievable costs are affected by the ratio between c_2 and c_1. In order to do that, the holding cost is fixed at $c_1 = 1$ \$/sec and the cost of running a server, c_2, is varied in the interval $0.04, \ldots, 25$ \$/sec. The total number of servers is 40 and the arrival rate is 70 jobs/sec, corresponding to an offered load of about 16.

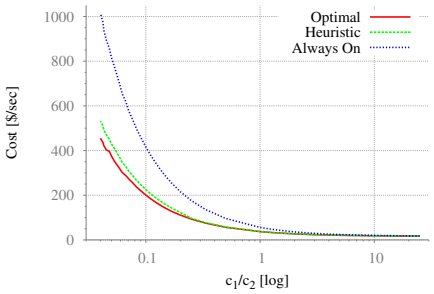


Figure 2: Cost for different values of c_2. $c_1 = 1.0$, $N = 40$, $\lambda = 70$.

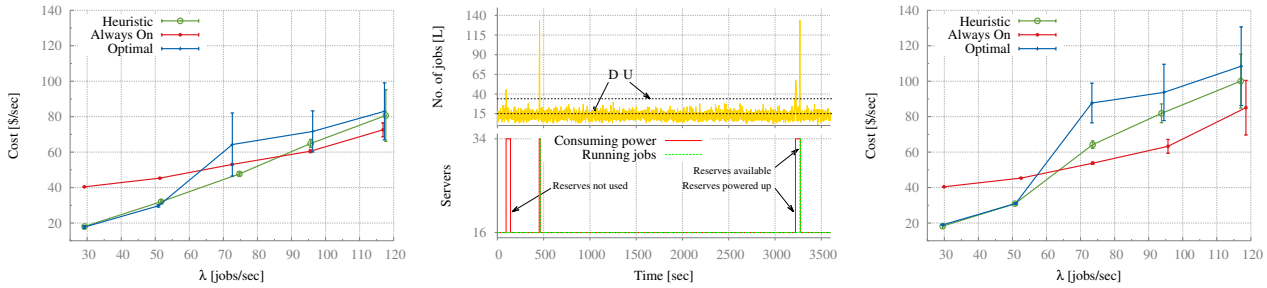
As one might expect, the difference between static and dynamic policies becomes smaller and smaller as the c_1/c_2 ratio increases: Figure 2 clearly shows that when the cost of running servers becomes negligible, the default policy is as good as the optimal one. Conversely, when $c_2 \gg c_1$ it becomes very important to choose the right number of servers to run as well as when to power the reserves on and off. For example, when $c_1/c_2 = 0.04$, the optimal cost is about 455 \$/sec, the cost of the default policy is over 1,016 \$/sec, while that of the heuristic is 532 \$/sec.

The following results are obtained from real-life experiments.

4.1 Non-bursty traffic

We evaluate the effectiveness of the dynamic allocation schemes on the Amazon EC2 cloud. For comparison reasons, the performance of the default policy which runs all the serves all the time is also displayed. We vary the load between approximately 20% and 80% by increasing the arrival rate from 29 to 118 jobs/sec. The interarrival intervals are exponentially distributed, *i.e.*, the input stream is Poisson. Each point in the figures below represent one run lasting one hour. During each run, roughly 115,000 (low load) to 400,000 jobs (high load) enter the system. These correspond to about 16 and 62 Mbits of traffic respectively being handled by the load balancer every second. Samples of the cost are collected every six minutes; they are used at the end of each run to compute the corresponding 95% confidence interval, which is calculated using the Sutdent's t-distribution.

The most notable feature of the graph plotted in Figure 3a is that the 'Heuristic' policy is practically indistinguishable from the more computationally intensive 'Optimal' under light load ($< 40\%$). Above that point the two differ, however the corresponding points are within each other's confidence intervals. The reason for that lies in the fact that the 'Heuristic' algorithm choses the parameters in a more conservative manner, hence over-provisioning the system in a larger measure compared to the 'Optimal' policy, while the two thresholds governing the reserve block are also smaller. As one might expect, as the load increases the difference between the dynamic policies and the 'Always On' approach becomes smaller and smaller. In fact, the results show that when c_1 is similar to c_2, employing a dynamic allocation scheme is advantageous only for loading conditions not exceeding 60–65%. Beyond that point the performance of the dynamic schemes worsens considerably, while the confidence intervals get much wider. It is perhaps worth noting that while running these experiments we have noticed that the cost function (1) is relatively robust with respect to the value of D and U – see also Figure 1 – but it is very sensitive with respect to λ. Due to the fact that the reserve block is powered on and off all the time, when the system is heavily loaded ($> 75\%$) even a relatively small under-estimation of the arrival rate ($\sim 2\%$) has a significant effect



(a) $ca^2 = 1$ (b) 'Heuristic' policy, $\lambda = 51$ (c) $ca^2 = 4$

Figure 3: Comparison between different policies: (a) Markovian traffic, (b) number of jobs in the system (top) and number of servers (bottom): heuristic policy, $\lambda = 51$, $ca^2 = 1$, $n = 16$, $D = 15$, $U = 34$, and (c) Lognormally distributed interarrival intervals, $ca^2 = 4$.

on the cost.

Figure 3b compares the number of jobs inside the system and the number of running servers, for the scenario where $\lambda = 51$ jobs/sec. As one can see, when a big burst of arrivals occurs, the reserves are powered on. In some circumstances (*e.g.*, first burst in the figure) they do not handle the extra load sufficiently quickly due to the power-up delay, and thus they are powered off as soon as they become available. In other circumstances, instead, they are used to deal with the unexpected load. Regarding this point, when reserves become available, they are used for a few seconds only, as the N servers deal with the backlog very quickly (the size of the queue will decrease at rate $N\mu - \lambda$). However if the load is extremely bursty it can be better to use the default policy which does not employ reserves, as it eliminates the unproductive power-up delays.

4.2 Bursty traffic

In the next experiments, the arrival process is no longer Poisson. The average arrival rate is kept the same as before, however now interarrival intervals are generated according to a Lognormal distribution.

One would expect that an increase in the interarrival time variability would increase the cost. Figure 3c, which illustrates the case where the squared coefficient of variation of the interarrival intervals is $ca^2 = 4$, shows that the reserve block model deals well with traffic bursts only for average loading conditions not exceeding 40% (the cost is the same as that of the Markovian scenario, see Figure 3a). If the load exceeds that threshold, however, it becomes better to employ the default policy, as the unproductive power-up delays increase the cost considerably. Also, the policy solving the queueing model performs worse than the 'Heuristic' algorithm due to the assumptions being violated (*i.e.*, the optimal solution is no longer optimal).

4.3 Time varying traffic

In real world scenarios the chances that service providers would have to deal with stationary traffic are extremely rare, while parameters are usually predicted and/or estimated at runtime because they are not known in advance. Hence, in the last experiment we study the case of time varying user demand. In oder to do so, we employ the trace of day 10 of the ClarkNet workload[1]. As far as the user demand forecasting is concerned, instead, we employ two different predictors, an 'Oracle' which has complete knowledge of the user demand, and a modified Holt-Winters' algorithm with multiplicative effects for both seasonal and error components, specifically the ETS(M,A,M) algorithm [6] as implemented by the R command `ets(ts, model="MAM", damped=F)`. Hence, we contrast three settings: 'Oracle' predictor paired with the 'Heuristic' policy, 'MAM' predictor paired with the 'Heuristic' policy, and 'MAM' predictor paired with a 'Static' policy, *i.e.*, a policy that runs the same amount of always on servers as the 'Heuristic' but does not employ the reserve block.

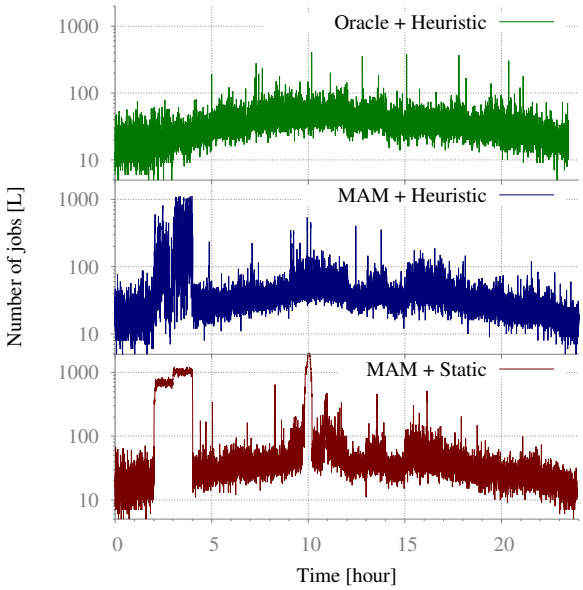


Figure 4: Number of jobs inside the system, L, for different policies. Every point represents the maximum value over a 5 seconds interval.

As shown in Figure 5a, the forecaster makes mistakes (*e.g.*, see hours 1–3) which have a negative impact on the cost. In particular, forecasting errors might cause overload situations, while our model requires the system to be stable. In order to avoid situations where the queue grows unbound HAProxy discards jobs that have been waiting for 30 seconds in the queue; nevertheless, the maximum queue size still grows to more than 1,000, see Figure 4.

Regarding the arrival rate, we use a Lognormal distribution where the mean value changes every hour in order to reflect the shape of the Clarknet traces, while the squared coefficient of variation is al-

[1] `http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html`

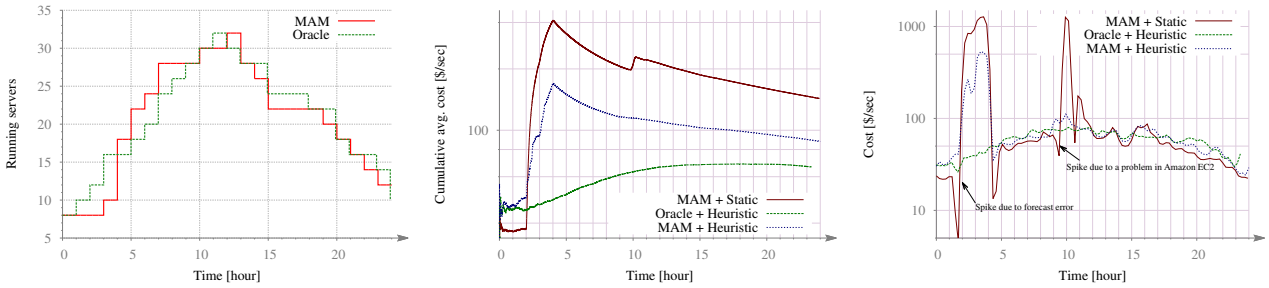


Figure 5: (a) Number of 'always on' servers, (b) cumulative average cost for different policies, and (c) cost per second for different policies. Data is collected with 1 second accuracy and lines are smoothed using a cubic polynomial.

ways equal to 4. The parameters governing the policies are also updated every hour. According to the logs collected by HAProxy during the 24 hours period, about 380 GB of data were served, corresponding to about 4.5 million jobs (the overall arrival rate is about 72 jobs/sec).

The most noticeable result is that forecasting errors are responsible for about a 45% increase in the cumulative average cost, see Figure 5b (58 $/sec versus 84 $/sec). In spite of prediction errors, however, employing the 'Heuristic' algorithm causes a large reduction in the cost compared to the case where no reserves are employed (84 $/sec versus 160 $/sec). This is due to the fact that, from an economic point of view, powering on/off the reserves very often is still more advantageous than leaving the jobs into the queue, see Figure 5c. That applies also to the case where the system is under-provisioned due to forecasting errors, see hours 2–4 in Figure 5c. Also, please note that idle periods occur even when the system is under-provisioned, see Figure 4. At the same time, Figure 4 also shows that due to the bursty traffic the 'Oracle' forecaster can not prevent temporary overload situations.

5 Conclusions

We have tested a dynamic power-saving policy using a main block of servers and a block of reserves both numerically and in a real-life setting. It has been shown that in normal operating conditions, when the arrival process is not excessively bursty, that policy can achieve significantly lower costs than the default policy where all servers are powered on all the time. Moreover, a simple and easily implementable heuristic produces results that are very close to, and sometimes better than those provided by an algorithm solving the corresponding queueing model. Only when the arrival process becomes extremely bursty or the load exceeds 60–65% is it better to employ the default policy.

When the number of always on servers can not deal with the average load, jobs experience large waiting times due to power up delays, while reserves are powered up and down very often. Hence, when the load is non-stationary, forecasting errors can have a huge impact on the performance of the system. In spite of that, however, our experiments have shown that the reserve-block model performs better that an algorithm that does not employ reserves to deal with temporary traffic spikes. As discussed, the load balancer employed a timeout in order to avoid situations where the queue grows unbound. Hence, a natural extension of the proposed model includes explicitly accounting for users' [im]patience. Another possible extension include considering multiple blocks of reserves, where each block is powered up and down at different thresholds. Finally, since the number of always-on servers plays a crucial role in determining the performance of our proposal, we plan to investigate allocation heuristics capable of better dealing with forecasting errors.

Acknowledgements

This work was partly funded by ERDF via the Estonian Competence Centre Programme, by the European Commission via the REMICS project (FP7-257793), and by the EU Cost Action IC0804.

6 References

[1] Artalejo, J. R., Economou, A., and Lopez-Herrero, M. J. Analysis of a multiserver queue with setup times. *Queueing Syst. Theory Appl. 51*, 1-2 (Oct. 2005), 53–76.

[2] Bobroff, N., Kochut, A., and Beaty, K. Dynamic placement of virtual machines for managing sla violations. In *IM '07* (May 2007), IEEE, pp. 119 –128.

[3] Bolla, R., Bruschi, R., Cianfrani, A., and Listanti, M. Enabling backbone networks to sleep. *Network, IEEE 25*, 2 (march-april 2011), 26 –31.

[4] Chase, J. S., Anderson, D. C., Thakar, P. N., Vahdat, A. M., and Doyle, R. P. Managing energy and server resources in hosting centers. *ACM SIGOPS Operating Systems Review 35*, 5 (2001), 103–116.

[5] Gandhi, A., Gupta, V., Harchol-Balter, M., and Kozuch, M. Optimality Analysis of Energy-Performance Trade-off for Server Farm Management. In *Performance* (November 2010).

[6] Hyndman, R., Koehler, A., Ord, J., and Snyder, R. *Forecasting with Exponential Smoothing – The State Space Approach.* Springer, 2008.

[7] Kumar, S., Talwar, V., Kumar, V., Ranganathan, P., and Schwan, K. vManage: loosely coupled platform and virtualization management in data centers. In *ICAC '09* (2009), ACM, pp. 127–136.

[8] Le Sueur, E., and Heiser, G. Dynamic Voltage and Frequency Scaling: the Laws of Diminishing Returns. In *HotPower'10* (2010), USENIX Association.

[9] Mazzucco, M., and Dyachuk, D. Optimizing cloud providers revenues via energy efficient server allocation. *Sustainable Computing: Informatics and Systems 2*, 1 (2012), 1 – 12.

[10] Mazzucco, M., Vasar, M., and Dumas, M. Squeezing out the Cloud via Profit-Maximizing Resource Allocation Policies. In *MASCOTS 2012* (August 2012), IEEE.

[11] Mitrani, I. Managing performance and power consumption in a server farm. *Annals of Operations Research* (2011).

[12] Schwartz, C., Pries, R., and Tran-Gia, P. A queuing analysis of an energy-saving mechanism in data centers. In *ICOIN 2012* (February 2012), pp. 70–75.

[13] Slegers, J., Thomas, N., and Mitrani, I. Dynamic server allocation for power and performance. In *SIPEW '08* (2008), Springer-Verlag, pp. 247–261.

[14] Urdaneta, G., Pierre, G., and van Steen, M. Wikipedia workload analysis for decentralized hosting. *Computer Networks 53*, 11 (2009), 1830–1845.

[15] Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P., and Wood, T. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst. 3*, 1 (Mar. 2008), 1:1–1:39.

[16] van Baaren, E.-J. Wikibench: A distributed, wikipedia based web application benchmark. Master's thesis, VU University Amsterdam, May 2009.

[17] Verma, A., Kumar, G., and Koller, R. The cost of reconfiguration in a cloud. In *Middleware* (2010), ACM, pp. 11–16.

[18] Zhang, Q., Cherkasova, L., Mi, N., and Smirni, E. A regression-based analytic model for capacity planning of multi-tier applications. *Cluster Computing 11*, 3 (Sept. 2008), 197–211.