MoKa: A System for Modeling and Capacity Planning of Multi-Tier Systems

Jean Arnaud, Sara Bouchenak

INRIA
RHÔNE-ALPES

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

MOKA: *A System for Modeling and Capacity Planning of Multi-Tier Systems*

Jean Arnaud — Sara Bouchenak

N° 6730

November 2008

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6730--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE – RHÔNE-ALPES

MoKa: A System for Modeling and Capacity Planning of Multi-Tier Systems

Jean Arnaud*, Sara Bouchenak†

Thème COM — Systèmes communicants
Équipe-Projet SARDES

Rapport de recherche n° 6730 — November 2008 — 33 pages

Abstract: Although cluster-based multi-tier data centers provide a means for supporting scalable web applications, their ad-hoc configuration poses significant challenges to the performance and economical costs of multi-tier applications. This paper presents the design and implementation of MoKa- a utility-aware framework for modeling multi-tier data centers and planning their capacity and optimal configuration. The contribution of the paper is threefold. First, we identify two levels of configuration of cluster-based multi-tier data centers, *local configuration* that applies at server's level and *architectural configuration* that relates to the clusters of servers in a multi-tier architecture. The combination of these two levels of configuration improves the overall performance and cost of cluster-based multi-tier data centers. Second, we present a *utility function* for characterizing the impact of local and architectural configurations on the performance and cost of multi-tier systems. Third, we develop a utility-aware capacity planning algorithm for efficiently calculating the optimal local and architecural configuration of multi-tier data centers to provide guarantees on performance while minimizing the cost. Our experiments on a multi-tier e-commerce auction site show the effectiveness of MoKa. Moreover, the experiments show that the combination of local and architectural configurations provides a 100% accurate utility for the multi-tier system, while with a single level of optimization (local or architectural) accuracy is limited between 20% and 90%.

Key-words: Multi-tier systems, Data centers, Modeling, Capacity planning, QoS, Optimization.

* INRIA
† Univ. Grenoble I – INRIA

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Contents

1 Introduction

Data centers host a large variety of Internet services, ranging from web servers to email servers, streaming media services, enterprise servers, and database systems [2, 21, 3, 17, 27]. These services are usually based on the client-server architecture, in which a server provides some online service to concurrent clients, such as reading web documents, sending emails or buying the content of a shopping cart. To face the increasing load of such applications, servers are organized in a multi-tier architecture. Figure 1 represents a three-tier web application which starts with requests from web clients that flow through an HTTP front-end server and provider of static content, then to an enterprise server to execute the business logic of the application and generate web pages on-the-fly, and finally to a database that stores non-ephemeral data. However, the complexity of multi-tier applications and their low rate for delivering dynamic web documents – often one or two orders of magnitudes slower than static documents – place a significant burden on data centers [15]. To face high loads and provide higher service scalability, a commonly used approach is the clustering and replication of servers in clusters of machines [25, 28, 7, 23].

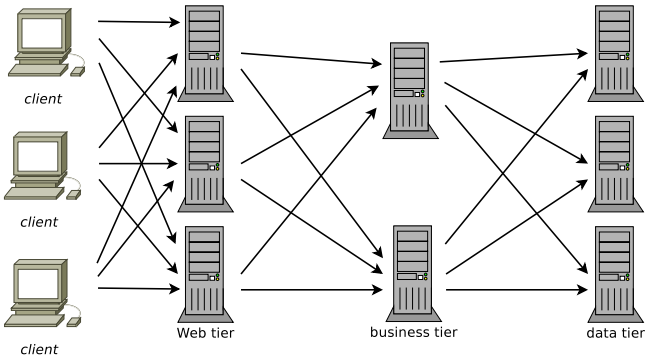


Figure 1: Multi-tier applications

The challenge in cluster-based multi-tier data centers stems from the conflicting goals of high performance and low cost and resource consumption. In the limit, high performance can be achieved by using all available resources in a data center to handle client requests. Symmetrically, it is possible to build a very-low cost data center by allocating very few resources which induces very bad performance and data center downtime [18]. Between these two extremes, there exists a configuration such that cluster-based multi-tier data centers achieve a desirable level of performance while cost is minimized. This paper precisely addresses the problem of determining this optimal configuration.

The contributions of the paper are the following:

- We identify and combine two levels of configuration of cluster-based multi-tier data centers: *architectural configuration* based on server provisioning in clusters in multi-tier data centers, and *local configuration* applied at server level. As far as we know, existing approaches are limited to a single level of configuration of multi-tier systems [31, 20, 13, 29]. In this paper

we show that unlike existing approaches, a two-level configuration allows guarantying performance constraints while minimizing the cost of multi-tier systems.

- We propose a *modeling* of cluster-based multi-tier systems that predicts the performance and cost of these systems. Our model is based on a queuing network that extends the MVA (Mean-Value Analysis) algorithm [26].
- We define a *utility function* for characterizing the impact of local and architectural configurations on the performance and cost of systems multi-tier systems.
- We develop a *utility-aware capacity planning algorithm* for efficiently calculating optimal local and architecural configuration of cluster-based multi-tier systems.

In addition to the above contributions, the paper presents the design and implementation of MoKa, a utility-aware framework for modeling cluster-based multi-tier data centers and planning their capacity and optimal configuration. Finally, the paper describes the evaluation of the proposed capacity planning method using a realistic cluster-based multi-tier auction site à-la *www.eBay.com*, compares it to standard approaches for capacity planning of multi-tier systems, and shows that the proposed method significantly improves the performance metrics and system cost. The results of our experiments show that compared to standard methods of capacity planning of multi-tier systems where 15% of data center resources may be wasted and 94% of total client requests may be rejected because of site overload, a capacity planning method that combines local and architectural configuration allows guarantying application performance and minimizing resource usage.

The remainder of the paper is organized as follows. Section 2 defines the underlying system model and background. Section 3 derives the proposed analytic model of cluster-based multi-tier data centers. Section 4 describes our capacity planning method. Section 5 presents the implemented MoKa prototype. Section 6 discuss the evaluation results. Section 7 provides an overview of related work. Section 8 draws our conclusions.

2 System Model and Background

2.1 System model

Multi-tier applications follow the client-server architecture where clients connect to a multi-tier system. A multi-tier system is composed of a series of tiers $T_1, T_2, ..., T_M$. Each tier is tasked with a specific concern. For instance, the multi-tier system in Figure 1 consists of tier T_1 responsible of processing the application web content, tier T_2 responsible of the application business logic, and tier T_3 responsible of the storage of non-ephemeral data of the application.

A client request to a multi-tier system first accesses tier T_1 and then may flow through successive tiers $T_2, T_3, ..., T_M$. More precisely, when a request is processed by tier T_i either a response is returned to tier T_{i-1} (or to the client if $i = 1$), or a subsequent request is sent to tier T_{i+1} (if $i < M$).

In its basic form, each tier consists of a single server. Multiple clients may concurrently access the same server. To prevent a server from thrashing when the number of concurrent clients grows, a classically used technique is admission control. It consists in fixing a limit for the maximum number of clients allowed to concurrently access a server – also known as the Multi-Programming Level (MPL) configuration parameter of servers; above this limit, incoming clients are abandoned (i.e. rejected). Thus, a client request processed by a multi-tier system either terminates successfully with a response to the client, or is abandoned because of a server's concurrency limit. Furthermore, the number of clients N (or workload) that try to concurrently access a multi-tier system may vary over time. This correponds to different client behaviors at different times, e.g. an e-mail service usually faces a higher workload in the morning than in the rest of day.

A server in a multi-tier data center is hosted by a resource (i.e. machine, node), and a resource is exclusively owned by a server. However, for scalability purposes, a tier is usually provisioned with multiple servers in a cluster built atop replication, partitioning and load balancing techniques; in the following, we consider fair load balancing techniques. If not provisioned adequately, cluster-based multi-tier applications may face a bottleneck on one of the tiers; but bottleneck does not occur simultaneously on multiple tiers at the same time as shown in this study [9]. Obviously, the more resources are assigned to cluster-based multi-tier systems, the more performance is improved. In the following, a cluster typically consists of tens to hundreds of servers, and servers inside a cluster are homogeneous in the sense that they have the same hardware architecture as it is typically the case in computer clusters of that size [8].

2.2 Performance and cost

Among the key metrics of interest for quantifying the Quality-of-Service (QoS) of multi-tier applications, we can cite:

Client request latency, defined as the necessary time to process a client request by the multi-tier system. It corresponds to the time duration between the time where a client sends a request to the system and the time where the client receives a response to that request. The average client request latency (or latency, for short) is denoted as ℓ. A low latency is a desirable behavior which reflects a reactive multi-tier system.

Client request abandon rate, defined as the ratio of requests that are abandoned compared to the total number of requests received by a multi-tier system. It is denoted as α. A low client request abandon rate (or abandon rate, for short) is a desirable behavior which reflects the level of availability of a multi-tier system.

SLA – Service Level Agreement – is a contract negotiated between clients and their service provider [19]. It specifies service level objectives (SLOs) for the performance that the service must guaranty, like the maximum latency ℓ_{max} and the maximum abandon rate α_{max} of multi-tier applications. It is important to notice that the combination of latency and abandon rate SLOs is important. Otherwise, providing guarantees solely on latency, for instance, may lead to a situation where the maximum latency objective only holds for 10% of client requests, while 90% of client requests are rejected due to the absence

of guarantee on abandon rate. In this paper, we show how to combine both performance objectives.

Besides performance levels that multi-tier systems must guaranty, the cost of the system is another aspect that is taken into account when provisioning multi-tier data centers.

Cost of multi-tier systems refers to the economical and energetical costs of these systems. It is directly related to the total number of resources that host a cluster-based multi-tier application, and is denoted as ω. Obviously, a low cost of a multi-tier system is a desirable behavior since it minimizes the maintenance cost of the system.

2.3 System configuration

We define the configuration κ of a cluster-based multi-tier system consisting of M tiers as a combination of architectural configuration and local configuration: $\kappa < AC, LC >$. In this paper, the *architectural configuration* of a multi-tier system is conceptualized as an array $AC < AC_1, AC_2, ..., AC_M >$, where AC_i is the number of resources (i.e. machines) at tier T_i of the multi-tier system. The *local configuration* of a multi-tier system is conceptualized as an array $LC < LC_1, LC_2, ..., LC_M >$, where LC_i is servers MPL (multi-programmig level) at tier T_i of the multi-tier system. For instance, the architectural configuration of the cluster-based three-tier system represented by Figure 1 is $AC < 3, 2, 3 >$ and the local configuration could be $LC < 200, 160, 100 >$ though not illustrated in the figure.

3 Modeling Multi-Tier Applications

We propose an analytic model that evaluates the performance and cost of cluster-based multi-tier systems. The system is modeled as a closed-loop queuing network based on the Mean-time Value Analysis (MVA) algorithm [26], and builds atop the model proposed in [29]. In addition to the application workload N (i.e. number of concurrent clients) and the number of tiers M of the multi-tier system, we extended th inputs of the model with the local and architectural configuration κ of the multi-tier system. Symmetrically, in addition to latency ℓ, we extended the original model with new outputs such as abadon rate $\alpha$and cost ω of the cluster-based multi-tier system.

Algorithm 1 derives the proposed model. The calculation of average latency of client requests is mainly based on the Mean-time Value Analysis (MVA) algorithm [26] and is inspired from [29], see lines 24-34 in Algorithm 1. The only difference is the integration of the cluster-based dimension where a tier may consist of multiple servers

Algorithm 1 derives the proposed model. in addition to inputs, the model is parametrized for a multi-tier application with the following: the number M of tiers of the multi-tier application, the application client think time Z, the visit ratios $V < V_1, V_2, ..., V_M >$ and the service times $S < S_1, S_2, ..., S_M >$ [29]. The client *think time* Z is the average time between the reception of a response by a client and the sending of the next request by that client. The *visit ratios* reflect the effect of client requests on application tiers. When a client request enters a multi-tier application at tier T_1, the request may generate sub-sequent

Algorithm 1: MO: Modeling of multi-tier systems

Input:
N : $\#clients$ $(i.e.\ workload)$
$\kappa < AC, LC >$: $multi-tier\ system\ configuration$
Output:
ℓ: $latency$
α: $abandon\ rate$
ω: $cost$ $(i.e.\ \#resources)$
Parameters:
M: $\#tiers$
Z: $client\ think\ time$
$V < V_1, V_2, ..., V_M >$: $visit\ ratios$
$S < S_1, S_2, ..., S_M >$: $service\ times$
Initialization:
$R_0 = Z$; $\tau = 0$; $\omega = 0$;

/* abandon rate */
for $m = 1; m \leq M; i++$ **do**
  if $m == 1$ **then**
    $wa_m = N$; /* workload trying to access the system */
  else
    $wa_m = we_{m-1} \cdot Min(1, \frac{V_m}{V_{m-1}})$;
  $we_m = Min(wa_m, LC_m \cdot AC_m)$; /*workload entering the system*/
for $m = M; m > 0; m--$ **do**
  if $m == M$ **then**
    $ar_m = Max(wa_m - we_m, 0)$;
  else
    $ar_m = Max(wa_m - we_m, 0) + wa_{m+1} \cdot ar_{m+1} \cdot \frac{V_m}{V_{m+1}}$;
  $ar_m = ar_m / wa_m$; /* abandon rate at each tier */
$\alpha = ar_1$; /* global abandon rate */
$N' = N \cdot (1 - \alpha)$; /* non-rejectend clients */

/* latency */
for $m = 1; m \leq M; m++$ **do**
  $Ql_m = 0$;
  $D_m = V_m \cdot S_m / AC_m$; /* service demand */
for $n = 1; n \leq N'; n++$ **do**
  for $m = 1; m \leq M; m++$ **do**
    $R_m = D_m \cdot (1 + Ql_m)$;
  $\tau = (\frac{n}{R_0 + \sum_{m=1}^{M} R_m})$; /* throughput */
  for $m = 1; m < M; m++$ **do**
    $Ql_m = \tau \cdot R_m$; /* Little's law */
$\ell = \sum_{m=1}^{M} R_m$; /* latency */
/* cost */
for $m = 1; m \leq M; m++$ **do**
  $\omega = \omega + AC_m$; /* total #resources */

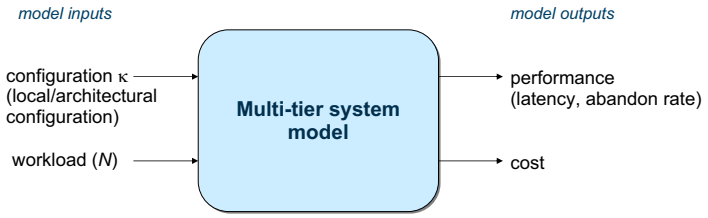


Figure 2: Modeling of multi-tier applications

requests to tier T_2, and then to tier T_3, and so on until tier T_M. The visit ratio V_i is the average number of sub-sequent requests at tier T_i generated by a client request entering the multi-tier application. The *service times* correspond to the incompressible amount of time necessary to process requests on application tiers. Thus, the service time S_i corresponds to the average time that is necessary to process a client request on tier T_i. In summary, these parameters reflect the behavior of a multi-tier application, and need to be identified for each multi-tier application through application profiling.

Algorithm 1 calculates latency, abandon rate and cost of a cluster-based multi-tier application. Latency calculation is mainly inspired from [29], with an extension from single server multi-tier systems to cluster-based multi-tier systems (cf. lines 23-33). The proposed algorithm also estimates the abandon rate of the multi-tier system, by applying admission control techniques based on MPL-related local configuration of servers in the multi-tier system (cf. lines 8-22). First, among the workload that tries to access each tier T_m of the multi-tier system (denoted as wa_m), the workload that actually enters the tier is calculated based on admission control (and denoted as we_m). Then, the abandon rate is calculated successively among each tierT_m (denoted as ar_m), until building the global abandon rate α. Finally, the algorithm calculates the cost of the cluster-based multi-tier system as total number of resources used by the system (cf. lines 34-36).

Algorithm cost. The model complexity is $O(NM)$, where N is the number of clients entering the system, and M the number of tier of the system ($N >> M$).

4 Capacity Planning

4.1 Utility function

Given the SLA objectives maximum latency ℓ_{max} and maximum abandon rate α_{max} that a multi-tier application must guaranty, we define *Performance Preference* as follows:

$$PP(\ell, \alpha) = (\ell \leq \ell_{max}) \cdot (\alpha \leq \alpha_{max}) \tag{1}$$

where ℓ and α respectively indicate the actual latency and abandon rate resulting from a multi-tier application. Note that $\forall \ell, \forall \alpha, PP(\ell, \alpha) \in \{0, 1\}$, depending on whether Eq. 1 holds or not.

Based on performance preference and cost of multi-tier applications, we now define a utility function that combines both criteria as follows:

$$\Theta(\ell, \alpha, \omega) = \frac{M \cdot PP(\ell, \alpha)}{\omega} \tag{2}$$

where ω is the actual cost (i.e. #resources) of the multi-tier application, and M is the number of tiers of the multi-tier application. M is used in Eq. 2 for normalization purposes. Here, $\forall \ell, \forall \alpha, \forall \omega, \Theta(\ell, \alpha, \omega) \in [0, 1]$, since $\omega \geq M$ (at least one server per tier) and $PP(\ell, \alpha) \in 0, 1$.

A high value of the utility function reflects the fact that, on the one hand, the performance of a multi-tier application guaranties service level objectives as specified by SLA, and on the other hand, the cost underlying the multi-tier application is low.

In order to illustrate the behavior of the utility function, a set of synthetic data for a three-tier application is given in Table 1 and Figure 3. Three workloads are considered, respectively 10 clients, 100 clients and 1000 clients. In Table 1 for each workload, different configurations κ_i of the multi-tier application are considered, varying at both architectural and local levels. Figure 3 gives for each configuration the corresponding value of utility function. For instance, with a workload of 1000 clients, the highest value of utility function is obtained with configuration κ_{16} which guaraties performance preference with a total cost of 9 resources. With the same workload, if the multi-tier application has a lower architectural configuration (i.e. less resources) as κ_{13}, or if it has a lower local configuration (i.e. lower MPL) as κ_{14}, performance preference for latency and abandon rate is no more guarantied. Symmetrically, if a higher architectural configuration is used as in κ_{17} an κ_{18}, this would increase the cost of the multi-tier application without any improvement on performance, with an overall decrease of utility. In case of a higher local configuration as in κ_{15}, the application faces higher concurrency which invalidates latency performance preference and thus decreases utility to 0. Similarly, with a workload of 100 clients and 10 clients, configurations that gauranty application performance preference with minimal cost are the ones that maximize the utility function, respectively κ_{10} and κ_2.

4.2 Utility-aware capacity planning

We propose a utility-aware capacity planning method that is based on the above utility function in order to calculate the *optimal* architectural and local config-

Workload	Configuration description
10 clients	$\kappa_1 : AC < 1,1,1 >, LC < 50,50,50 >$ $\kappa_2 : AC < 1,1,1 >, LC < 100,100,50 >$ $\kappa_3 : AC < 1,2,1 >, LC < 100,50,100 >$ $\kappa_4 : AC < 1,2,2 >, LC < 200,100,150 >$ $\kappa_5 :< AC < 2,2,3 >, LC < 200,200,100 >$ $\kappa_6 :< AC < 2,3,4 >, LC < 300,200,200 >$
100 clients	$\kappa_7 :< AC < 1,1,1 >, LC < 50,50,50 >$ $\kappa_8 :< AC < 1,2,2 >, LC < 50,50,50 >$ $\kappa_9 :< AC < 2,2,2 >, LC < 20,50,50 >$ $\kappa_{10} :< AC < 1,2,2 >, LC < 100,50,50 >$ $\kappa_{11} :< AC < 2,2,2 >, LC < 100,100,50 >$ $\kappa_{12} :< AC < 2,4,4 >, LC < 200,100,200 >$
1000 clients	$\kappa_{13} :< AC < 2,2,4 >, LC < 100,200,100 >$ $\kappa_{14} :< AC < 2,3,4 >, LC < 200,200,200 >$ $\kappa_{15} :< AC < 2,3,4 >, LC < 500,400,400 >$ $\kappa_{16} :< AC < 2,3,4 >, LC < 300,200,200 >$ $\kappa_{17} :< AC < 2,4,4 >, LC < 300,200,200 >$ $\kappa_{18} :< AC < 4,3,4 >, LC < 300,200,200 >$

Table 1: Configuration examples

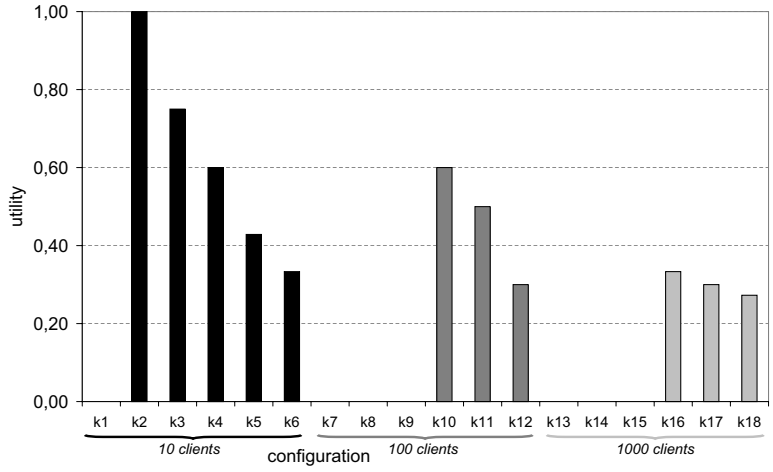


Figure 3: Utility function

uration of a cluster-based multi-tier application in such a way that the application performance preference for latency and abandon rate is guarantied while the cost of the application is minimized. Calculating the optimal configuration of a multi-tier application is thus equivalent to calculating the configuration for which the utility function value is maximal, (i.e. optimal, Θ^*).

Algorithm 2 describes our capacity planning algorithm for cluster-based multi-tier applications. The overall behavior of the algorithm is presented in Figure 4. Roughly speaking, given an application workload and a target performance preference in terms of maximum latency and abandon rate, a capacity planning algorithm produces an intial minimal configuration of the multi-tier application, then calculates the performance of this configuration based on the multi-tier application model, and tests the produced performance against performance preference. If performance preference is verified, the capacity planning algorithm returns that configuration and terminates; otherwise, it iterates on

Algorithm 2: KA: Capacity planning of multi-tier systems - Tree-based search

Input:
N : $\#clients$ $(i.e.\ workload)$
Output:
$\kappa^* < AC, LC >$: $multi-tier\ system\ configuration$
Parameters:
ℓ_{max} : $maximum\ latency$
α_{max} : $maximum\ abandon\ rate$
M : $\#tiers$
$MPL_{max} < MPL_{max-1}, ..., MPL_{max-M} >$: $maximum\ MPL\ at\ each\ tier$
$V < V_1, V_2, ..., V_M >$: $visit\ ratios$
MO : $model\ of\ the\ multi-tier\ system$
Initialization:
/* $ignore\ \alpha_{max}\ incoming\ clients$ */
$N' = N * (1- \alpha_{max})$;
$\alpha'_{max} = 0$;
/* $initial\ architectural\ and\ local\ configuration$ */
for $m = 1; m \leq M; m++$ **do**
  $AC_m = 1$;
  $LC_m = N' \cdot V_m$;
$o^* = 0$;
$\kappa^* =< \emptyset, \emptyset >$;
/* $AC\ and\ LC\ that\ verify\ \ell_{max}\ and\ \alpha'_{max}\ conditions$ */
while $\ell > \ell_{max} \vee \alpha > \alpha'_{max}$ **do**
  for $m = 1; m \leq M; m++$ **do**
    $AC_m = AC_m + 1$;
    $LC_m = MIN(\frac{N' \cdot V_m}{AC_m}, MPL_{max-m})$;
  $< \ell, \alpha, \omega > = MO(N', \kappa < AC, LC >)$;
/* $AC\ cost\ minimization$ */
for $m = 1; m \leq M; m++$ **do**
  /*$dichotomic\ search\ of\ AC^*_m\ in\ [1 \ldots AC_m]$ */
  $AC'_m = \frac{AC_m}{2}$;
  $LC'_m = MIN(\frac{N' \cdot V_m}{AC'_m}, MPL_{max-m})$ $AC' =< AC_1, ..., AC'_m, ..., AC_M >$;
  $LC' =< LC_1, ..., LC'_m, ..., LC_M >$;
  $< \ell, \alpha, \omega >= MO(N', \kappa < AC', LC' >)$;
  if $\ell > \ell_{max} \vee \alpha > \alpha'_{max}$ **then**
    $pursue\ dichotomic\ search\ of\ AC^*_m\ in] \frac{AC_m}{2} \ldots AC_m]$
  else
    $pursue\ dichotomic\ search\ of\ AC^*_m\ in\ [1 \ldots \frac{AC_m}{2}]$
/*$at\ the\ end$: $AC < AC^*_1, ..., AC^*_m, ..., AC^*_M >$*/
/* $LC\ improvement$ */
for $m = 1; m \leq M; m++$ **do**
  /* $dichotomic\ search\ of\ LC^*_m in\ [LC_m \ldots MPL_{max-m}]$ */
  $LC'_m = \frac{MPL_{max-m} - LC_m}{2} + LC_m$ $LC' =< LC_1, ..., LC'_m, ..., LC_M >$
  $< \ell, \alpha, \omega > = MO(N, \kappa < AC, LC' >)$
  if $\ell > \ell_{max}$ **then**
    $pursue\ dichotomic\ search\ of\ LC^*_m\ in\ [LC_m \ldots LC'_m[$
  else
    $pursue\ dichotomic\ search\ of\ LC^*_m\ in\ [LC'_m \ldots MPL_{max-m}]$
/*$at\ the\ end$: $LC < LC^*_1, ..., LC^*_m, ..., LC^*_M >$*/

a new augmented configuration of the multi-tier application and repeats the process.

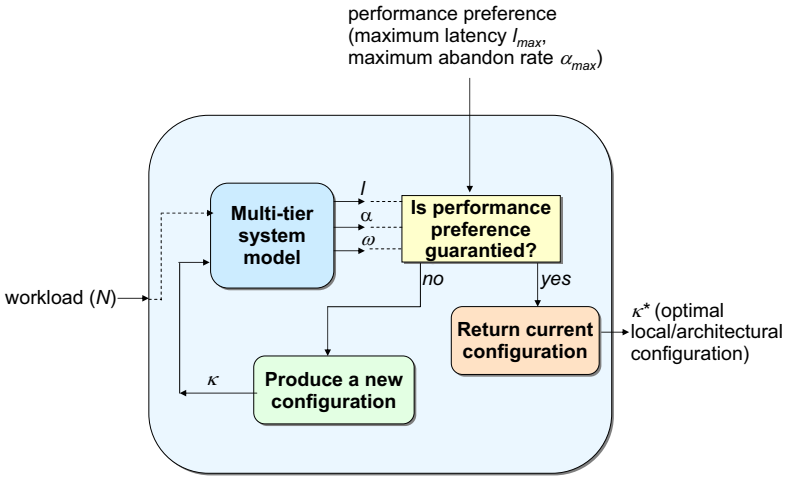


Figure 4: Capacity planning algorithm

Tree-based search. We propose a capacity planning algorithm for multi-tier applications in Algorithm 2. This algorithm is composed of three parts: first, it keep only a certain amount of entering workload, depending on the allowed abandon rate, then it process AC_{max}, the upper bound for the replication level at each tier, second it perform a dichotomic search into architectural configuration spaces, then in a third part it improves LC configuration. To determine AC_{max}, we process the utility function with an AC equal to 1 on each tier, then we add one resource on each tier until performance constraints are matched. Resulting AC_{max} will be the upper bound for the architectural configurations. The dichotomic search is used to find the optimal architectural configuration, i.e. the one that minimizes resources consumption while matching QoS constraints. The local configuration is adapted on each step to match the amount of admitted clients and allocated resources.

The third part aims to make our algorithm more robust to load variations. We perform a search, on each tier, for the highest value of the local configuration that still match QoS constraints. Using this algorithm, system reconfiguration can be avoided in case of small workload amount variations.

Algorithm cost. Our capacity planning algorithm complexity is

$$O(N.M.AC_{max} + N.M^2.(\log_2(AC_{max}) + \log_2(LC_{max}))) \quad (3)$$

where N is the number of clients entering the system, M the number of tier of the system ($N >> M$), AC_{max} the maximum replication level (on the bottleneck tier), and LC_{max} the maximum local configuration value.

Exhaustive search. A capacity planning algorithm based on an exhaustive search on all possible architectural and local configurations of a multi-tier application is described in Algorithm 3.

The principle is to build the set Φ_{AC} of all possible architectural configurations and the set Φ_{LC} of all possible local configurations. All the combinations of Φ_{AC} and Φ_{LC} are possible system configuration. We test them exhaustively to find the one that maximize our objective function, while minimizing resources consumption. In order to build the Φ_{AC} and Φ_{LC} sets, we have to limit the local and architectural configuration spaces. For the local configurations, we chosed the maximum possible value of the MPL, which depends on the underlying software used at each tier. Then to determine the maximum value of AC, we process the utility function with an AC equal to 1 on each tier, then we add one resource on each tier until performance constraints are matched. Resulting AC_{max} will be the upper bound for the architectural configurations.

Algorithm cost. The exhaustive search capacity planning algorithm complexity is

$$O(N.M.AC_{max} + N.M.(LC_{max}^{M} * AC_{max}^{M})) \tag{4}$$

where N is the number of clients entering the system, M the number of tier of the system ($N >> M$), AC_{max} the maximum replication level (on the bottleneck tier), and LC_{max} the maximum local configuration value.

4.3 Accuracy of the capacity planning

In this section, we prove the accuracy of our tree-based search capacity planning algorithm. To make the text easier to read, we use k to represent a configuration produced by our capacity planning planning algorithm, and $\Theta(k)$ to represent this configuration utility. (instead of $\Theta(\ell, \alpha, \omega)$, see 4.1).

Let k^* be the optimal configuration for a given set of constraints. We define the accuracy of a configuration k as follow:

$$acc(k) = \frac{\Theta(k)}{\Theta(k^*)} \tag{5}$$

There are three properties underlying this proof:

Property 1: There is only one bottleneck tier (or none) at a given moment.

Property 2: Adding more resources never degrades performance (and often improve performance)

Property 3: Constraints on latency and abandon rate must be realistic: they must be reachable by adding a certain amount of resources. As an exemple, you cannot specify a latency constraint inferior to request service times.

So we have to prove that all the configurations processed by our algorithm have an accuracy equal to 1, which is the maximum value. We will prove it by contradiction, assuming that there exists another configuration k' with $\Theta(k') > \Theta(k)$.

Algorithm 3: Capacity planning of multi-tier systems - Exhaustive search

Input:
$N : \#clients\ (i.e.\ workload)$
Output:
$\kappa^* < AC, LC >:\ multi-tier\ system\ configuration$
Parameters:
$\ell_{max}:\ maximum\ latency$
$\alpha_{max}:\ maximum\ abandon\ rate$
$M:\ \#tiers$
$MPL_{max} < MPL_{max-1}, ..., MPL_{max-M} >: maximum\ MPL\ at\ each\ tier$
$MO:\ model\ of\ the\ multi-tier\ system$
Initialization:
/ initial architectural and local configuration */*
for $m = 1; m \leq M; m++$ **do**
  $AC_m = 1;$
  $LC_m = MPL_{max-m};$
$o^* = 0;$
$\kappa^* =< \emptyset, \emptyset >;$
/ determine* AC_{max} *as* $MAX\{AC_i\}$ **/*
$< \ell, \alpha, \omega > = MO(N, \kappa < AC, LC >);$
while $\ell > \ell_{max} \vee \alpha > \alpha_{max}$ **do**
  for $m = 1; m \leq M; m++$ **do**
    $AC_m = AC_m + 1;$
  $< \ell, \alpha, \omega > = MO(N, \kappa < AC, LC >);$
$AC_{max} = AC_1;$
/$\Phi_{AC}:\ all\ possible\ architectural\ configurations$*/*
$\Phi_{AC} = \{AC < 1, ..., 1 >, ..., AC < AC_{max}, ..., AC_{max} >\}$
/ $|\Phi_{AC}| = (AC_{max})^M$ */*
/$\Phi_{LC}:\ all\ possible\ local\ configurations$*/*
$\Phi_{LC} = \{LC < 1, ..., 1 >, ..., AC < MPL_{max-1}, ..., MPL_{max-M} >\}$
/ $|\Phi_{LC}| = \prod_{m=1}^{M} MPL_{max-m}$ */*
foreach $AC \in \Phi_{AC}$ **do**
  foreach $LC \in \Phi_{LC}$ **do**
    $< \ell, \alpha, \omega > = MO(N, \kappa < AC, LC >)$
    if $\Theta(\ \ell, \alpha, \omega) > o^*$ **then**
      $o^* = \Theta(\ \ell, \alpha, \omega);$
      $\kappa^* = \kappa;$

$$\exists k', \Theta(k') > \Theta(k) \tag{6}$$

$$\Leftrightarrow \quad \exists k', \frac{M \cdot PP_{k'}(\ell, \alpha)}{\omega_{k'}} > \frac{M \cdot PP_k(\ell, \alpha)}{\omega_k} \tag{7}$$

Assuming that constraints can always be matched (Property 3), $PP_k(\ell, \alpha) = 1$ and $PP_{k'}(\ell, \alpha) = 1$. Then:

$$\exists k', \frac{1}{\omega_{k'}} > \frac{1}{\omega_k} \tag{8}$$

$$\Leftrightarrow \quad \exists k', \omega_{k'} < \omega_k \tag{9}$$

$$\Leftrightarrow \quad \exists k', \exists i \in 1; M, AC'_i < AC_i \tag{10}$$

where AC and AC' are the architectural configurations of k and k', respectively. Since the total cost of k' is lower than k cost, there is at least on tier with a lower replication level in k' than in k. We define k_{max} and k'_{max} the maximum replication level of k and k' respectively. k_{max} and k'_{max} are processed at the beginning of our algorithm to bound architectural configuration space.

First case. In this first case, we assume that $k'_{max} > k_{max}$, i.e. there is at least a tier i where $AC'_i < AC_i$ and a tier j where $AC'_j > AC_j$.

The first part of our algorithm, which set k_{max}, stop when QoS constraints are matched with an architectural configuration of $< k_{max} \dots k_{max} >$. If our algorithm stop with a given k_{max} value, performances constraints are matched with this configuration, and there is no need to add more resources on any tier (Property 1). Thus this case can't happen, and we fall back on the second case.

Second case. Now, we assume that $k_{max} = k'_{max}$, then k and k' have the same maximum replication level. There exists a tier i where $AC'_i < AC_i$, but due to property 2 and the dichotomic search construction, at least AC_i resources are required on tier i to match QoS constraints. ($\forall AC'_i, AC'_i < AC_i \Longrightarrow PP(AC'_i) = 0$). Since we assumed that performance constraints are matched with k', k' can not exist and k has an accuracy value of 1, whatever the context.

5 Implementation Details of MoKa

5.1 MoKa prototype

We designed and implemented the MoKa framework for modeling and capacity planning of multi-tier applications. The MoKa prototype integrates the implementation of the proposed analytical model (cf. Section 3), and capacity planning algorithm (cf. section 4). Furthermore, MoKa is an open framework that is able to integrate and compare different performance models and capacity planning algorithms In particular, it is able to evaluate the *accuracy* and *efficiency* of performance models and capacity planning algorithms.

The *accuracy* of a *performance model* is the ratio between the performance as predicted by the model and the actual performance of the real system (i.e. accuracy of latency and accuracy ofabandon rate).

The *efficiency* of a *performance model* is the necessary time for the model to predict the performance of the multi-tier system.

The *accuracy* of a *capacity planning algorithm* is the ratio between the result of the objective function when applying this capacity planning algorithm and the result of the objective function when applying the optimal capacity planning algorithm (based on an exhaustive search).

The *efficiency* of a *capacity planning algorithm* is the necessary time for the algorithm to calculate the configuration of the multi-tier system.

User interface. Moreover, we developed a user web interface for MoKa to allow users and administrators to remotely run MoKa on a web browser and calculate architectural and local configurations of multi-tier applications [1]. This is a useful tool for system administrators: they can enter their system parameters, workload informations, and MoKa will plot the evolution of the optimal (then suggested) local and architectural configurations when load increase.

Figure 5 shows the user interface of MoKa.

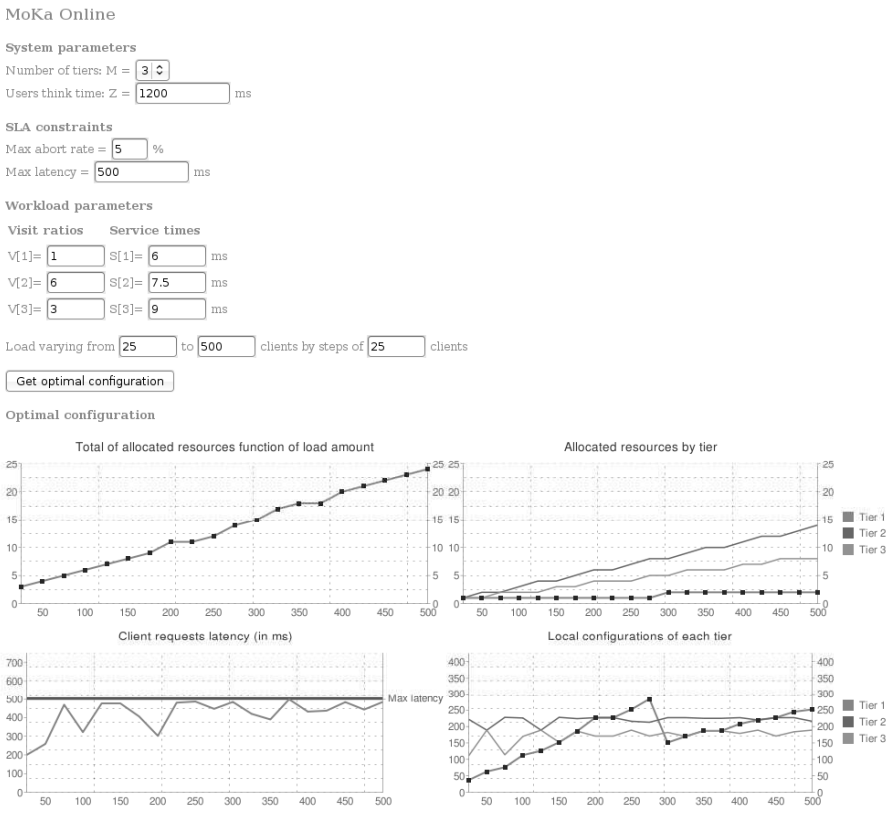


Figure 5: Screenshot of MoKa online

[1] *http://sardes.inrialpes.fr/research/moka/*

5.2 MoKa organization

The MoKa framework is designed to be as modular as possible. Thus it is made of 5 different packages, which are summarized in figure 6 :

- The *emodeling* package contains all the models of the underlying system
- The *kplanning* package contains all the capacity planning algorithms. Since capacity planning algorithms use models of the system, this package depends on the previous one
- The *gui* package contains classes for user interface, i.e. applet and servlet
- The *test* package contains a batch of automated tests to compare algorithm and model behavior and performance. Then it depends on emodeling and kplanning packages
- Then the *util* package embeds few utility classes common to emodeling or kplanning packages

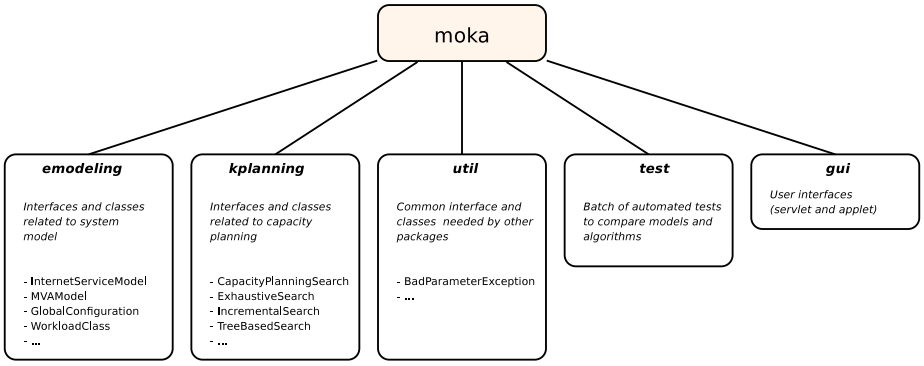


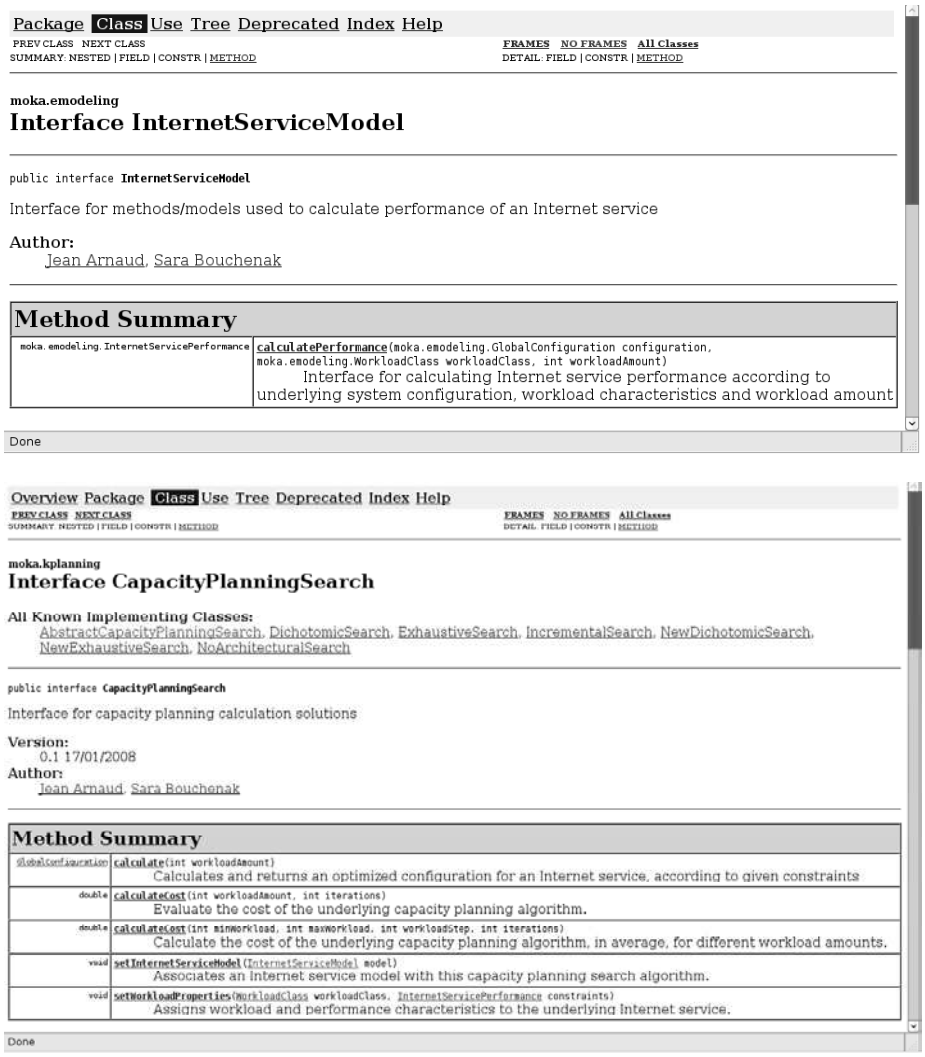
Figure 6: MoKa packages organization

Now we have described the general structure of the MoKa framework, let's go into details about the two main packages, respectively *emodeling* and *kplanning*. First, the *emodeling* package is in charge of processing some performance characteristics (*InternetServicePerformance*) in fonction of system configuration (*GlobalConfiguration*) and workload class (*WorkloadClass*) applied to the system. Since we can use different system models, we abstracted the performance calculation method into an interface (*InternetServiceModel*) which is implemented by the different models (*MVAModel*, ...).

The InternetServiceModel interface is given on figure 7. Each model of the framework has only to implements one method to be integrated to the framework.

Second, the *kplanning* package, which structure is given in figure 8, uses Internet service models to calculate an optimized configuration.

The structure of the *CapacityPlanningSearch* interface is given in Figure 7. After having set the model to use with *setInternetServiceModel*, we can specify workload properties and QoS constraints using *setWorkloadProperties*. Then



Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS
SUMMARY: NESTED | FIELD | CONSTR | METHOD

FRAMES NO FRAMES All Classes
DETAIL: FIELD | CONSTR | METHOD

moka.emodeling
Interface InternetServiceModel

public interface **InternetServiceModel**

Interface for methods/models used to calculate performance of an Internet service

Author:
Jean Arnaud, Sara Bouchenak

Method Summary

moka.emodeling.InternetServicePerformance	**calculatePerformance**(moka.emodeling.GlobalConfiguration configuration, moka.emodeling.WorkloadClass workloadClass, int workloadAmount) Interface for calculating Internet service performance according to underlying system configuration, workload characteristics and workload amount

Done

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS
SUMMARY: NESTED | FIELD | CONSTR | METHOD

FRAMES NO FRAMES All Classes
DETAIL: FIELD | CONSTR | METHOD

moka.kplanning
Interface CapacityPlanningSearch

All Known Implementing Classes:
AbstractCapacityPlanningSearch, DichotomicSearch, ExhaustiveSearch, IncrementalSearch, NewDichotomicSearch, NewExhaustiveSearch, NoArchitecturalSearch

public interface **CapacityPlanningSearch**

Interface for capacity planning calculation solutions

Version:
0.1 17/01/2008
Author:
Jean Arnaud, Sara Bouchenak

Method Summary

GlobalConfiguration	**calculate**(int workloadAmount) Calculates and returns an optimized configuration for an Internet service, according to given constraints
double	**calculateCost**(int workloadAmount, int iterations) Evaluate the cost of the underlying capacity planning algorithm.
double	**calculateCost**(int minWorkload, int maxWorkload, int workloadStep, int iterations) Calculate the cost of the underlying capacity planning algorithm, in average, for different workload amounts.
void	**setInternetServiceModel**(InternetServiceModel model) Associates an Internet service model with this capacity planning search algorithm.
void	**setWorkloadProperties**(WorkloadClass workloadClass, InternetServicePerformance constraints) Assigns workload and performance characteristics to the underlying Internet service.

Done

Figure 7: Javadoc for InternetServiceModel and CapacityPlanningSearch interfaces

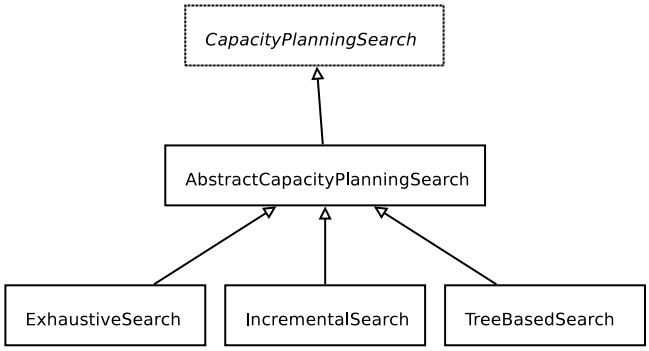


Figure 8: MoKa kplanning package structure

we can use the *calculate* method to get a GlobalConfiguration optimized for the specified workload amount.

6 Evaluation

6.1 Evaluation environment

Hardware environment. The experiments were conducted on a cluster of x86-compatible machines with bi-2.2GHz AMD Opteron CPUs and 4 GB RAM, connected via a 10 Gb/s Ethernet LAN.

Software environment. Our experiments run on the Linux 2.6.18 kernel and on the following middleware implementations: Apache Tomcat 5.5.23 Web and enterprise server [28], MySQL 5.0.37 database server [27], PLB 0.3 as the web server clustering solution [24], and Sequoia 2.10.6 load balancer [11].

Applications. We used two web applications as a basis of our experiments on MoKa, an auction site and the 1998 Soccer World Cup web site.

The *auction site* is based on Rubis, a J2EE multi-tier application benchmark [1]. Rubis defines several web interactions (e.g registering new users, browsing, buying or selling items). It provides a benchmarking tool that emulates web client behavior and generates a tunable workload; this allows us to vary the workload during the experiments. The benchmarking tool gathers statistics about the application such as average client request latency. We used an improved version of Rubis 1.4.2 with the browsing mix, where we added new statistics such as client request abandon rate. In our experiments, the auction site was deployed as a cluster-based replicated multi-tier system, consisting of a cluster of replicated Web/enterprise servers as a front-end *and* a cluster of replicated database servers as a backend.

The *1998 World Cup Web site* workload was characterized in [5]. Measurements and real traces from this site were collected over a three month period, during which 1.35 billion client requests were received [4]. In our experiments, the World Cup Web site was deployed as a cluster-based replicated mono-tier

system, consisting of a cluster of replicated Web servers. Althrough this mono-tier application is a particular case of multi-tier systems, it has the advantage of representing real traces with a high and varying workload, which motivates the usefulness of calculating the best architectural and local configuration of the application.

Model and capacity planning calibration. The proposed performance model and capacity planning algorithm were calibrated with appropriate parameters through offline profiling of the auction site. The applied parameters of Algorithms 1, 2 and 3 are given in Table 2.

	Modeling	**Capacity planning**
Auction site	$M = 2$ $Z = 4\ s$ $V < 1, 4.32 >$ $S < 7.92\ ms, 19\ ms >$	$\ell_{max} = 1\ s$ $\alpha_{max} = 10\%$ $V < 1, 4.32 >$ $MPL_{max} < 1000, 1000 >$
World Cup Web site	$M = 1$ $Z = 4\ s$ $V < 1 >$ $S < 5\ ms >$	$\ell_{max} = 1\ s$ $\alpha_{max} = 5\%$ $V < 1 >$ $MPL_{max} < 1000 >$

Table 2: Calibration parameters

6.2 Accuracy of the the model

The aim of this experiment is to validate model accuracy, by comparing the model predictions with measures performed when running the auction site benchmark. The architectural configuration is $AC =< 1; 1 >$, and the load amount grows on each measurement point. Figure 9 shows the predicted and measured latency in function of workload amount. The average error between measures and prediction is below 12%.

6.3 Ad-hoc vs. optimized multi-tier data centers

In the following experiments, we first compare the behavior of multi-tier systems configured in an ad-hoc way with a fixed architectural and a fixed local configuration, with the behavior of optimized systems based on the proposed utility-aware capacity planning method. Here, capacity planning aims at determining the best architectural and local configuration of a cluster-based multi-tier system in such a way that 90% of client requests are handled in less than 1 s (see Table 2), and the total cost of the system is minimized.

Table 3 first describes the two ad-hoc configurations that we consider, namely $AA - AL_1$ and $AA - AL_2$, with an Ad-hoc Architectural configuration and an Ad-hoc Local configuration (the other configurations of the table are introduced in Section 6.4). $AA-AL_1$ and $AA-AL_2$ were chosen to represent respectively a small configuration (with few machines and limited concurrency on servers), and a large configuration. Moreover, in $AA-AL_1$ the local configuration is set with the default MPL values that come with the Tomcat front-end server software and the MySQL back-end server software, namely 200 and 100. $OA - OL$ represents the multi-tier system where both architectural and local configurations are optimized using our capacity planning method.

Figures 10 and 11 respectively present the variation of latency and abandon rate of the cluster-based multi-tier auction site when the workload varies. Here,

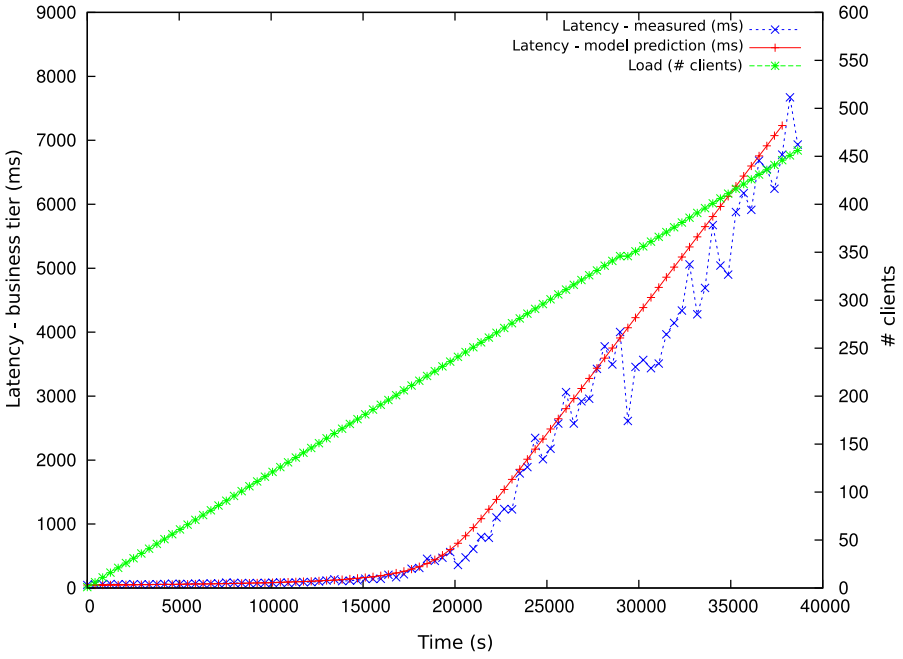


Figure 9: Model accuracy - Auction site

Auction site		
Configuration	**AC**	**LC**
$AA\text{-}AL_1$	Ad-hoc $AC < 1, 2 >$	Ad-hoc $LC < 200, 100 >$
$AA\text{-}AL_2$	Ad-hoc $AC < 6, 15 >$	Ad-hoc $LC < 300, 200 >$
$OA\text{-}OL$	Optimized	Optimized
$AA\text{-}OL_1$	Ad-hoc $AC < 1, 2 >$	Optimized
$AA\text{-}OL_2$	Ad-hoc $AC < 6, 15 >$	Optimized
$OA\text{-}AL_1$	Optimized	Ad-hoc $LC < 200, 100 >$
$OA\text{-}AL_2$	Optimized	Ad-hoc $LC < 300, 200 >$
World Cup site		
Configuration	**AC**	**LC**
$AA\text{-}AL_1$	Ad-hoc $AC < 10 >$	Ad-hoc $LC < 150 >$
$AA\text{-}AL_2$	Ad-hoc $AC < 30 >$	Ad-hoc $LC < 300 >$
$OA\text{-}OL$	Optimized	Optimized
$AA\text{-}OL_1$	Ad-hoc $AC < 10 >$	Optimized
$AA\text{-}OL_2$	Ad-hoc $AC < 30 >$	Optimized
$OA\text{-}AL_1$	Optimized	Ad-hoc $LC < 150 >$
$OA\text{-}AL_2$	Optimized	Ad-hoc $LC < 300 >$

Table 3: System configurations

$AA - AL_1$ is not able to guaranty the α_{max} abandon rate performance preference when the workload increases since it represents a small configuration (few machines and limited concurrency on servers). As a consequence of the limited concurrency, $AA - AL_1$ provides a low latency [2] that respects the ℓ_{max} preference. On the other hand, $AA - AL_2$ represents a large configuration (many machines and higher concurrency on servers). Thus, no requests are abandoned and all requests are concurrently executed on servers, with a direct impact on latency which dramatically grows and does not respect the ℓ_{max} latency preference when the workload increases. In contrast, the $OA - OL$ fully optimized multi-tier system obviously guaranties both α_{max} and ℓ_{max} performance preferences.

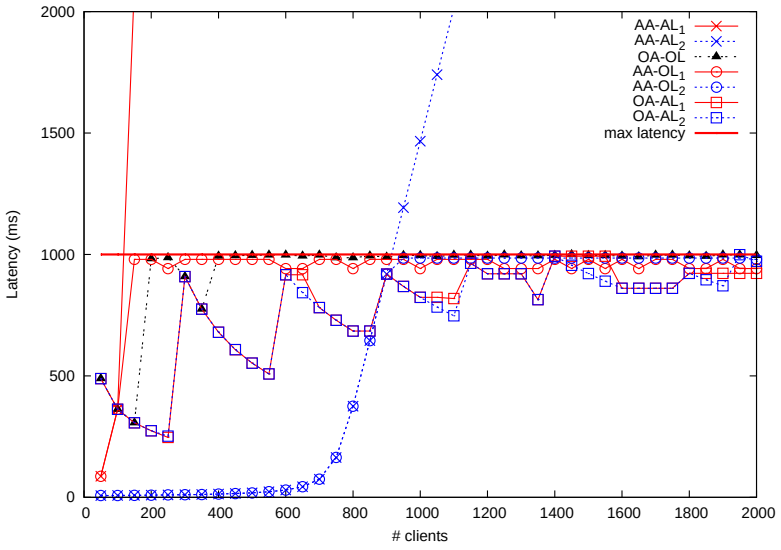


Figure 10: Auction site – Latency

[2] Latency of sucessfully terminated requests.

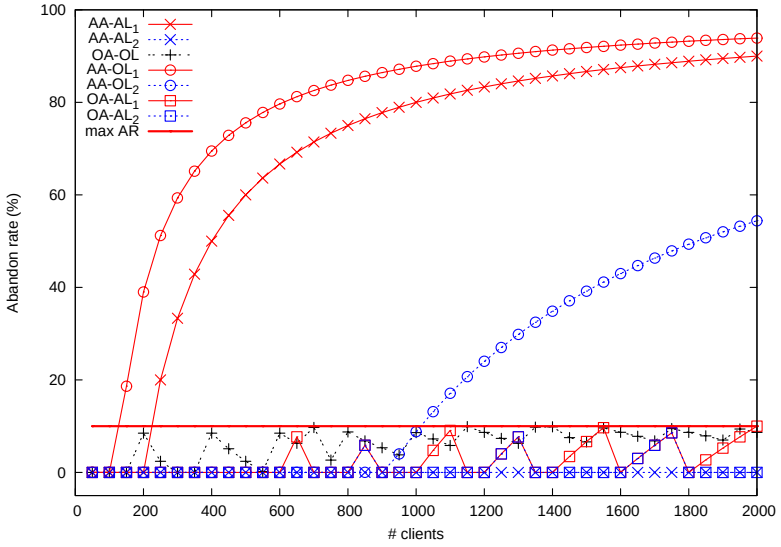


Figure 11: Auction site – Abandon rate

6.4 Local vs. architectural optimization

Previous works on capacity planning of multi-tier systems usually restrict their application to one level of configuration, either local configuration [13, 20, 31], or architectural configuration [29]. We argue that combining both levels of configuration improves the global behavior of the system and provides combined guarantees on multiple performance preferences while economizing resources. In the following, we compare multi-tier systems uniquely optimized with regard to their local configuration, systems uniquely optimized with regard to their architectural configuration, with systems where optimization applies at both levels of configuration.

In addition to the previously presented configurations, Table 3 describes four additional configurations [3] , namely $AA - OL_1$, $AA - OL_2$, $OA - AL_1$ and $OA - AL_2$. $AA - OL_1$, $AA - OL_2$ represent two configurations with an ad-hoc architectural configuration and an optimized local configuration based on classical admission control similar to the one presented in [20]. $OA - AL_1$ and $OA - AL_2$ represent two configurations where the local configuration is set in an ad-hoc way and the architecture is optimized following an approach similar to [29]. Figure 10 shows that $AA - OL_1$ and $AA - OL_2$ verify the ℓ_{max} latency performance preference because they are able to optimize their local configuration by limiting concurrency on servers. However, this has a direct impact on the abandon rate which grows when the workload increases; and thus does not respect the α_{max} preference as shown in Figure 11. The abandon rate grows later for $AA - OL_2$ than for $AA - OL_1$ because the former represents a larger configuration with more resources that temporarily absorb the workload. In contrast, $OA - OL$ is an architecturally and locally optimized configuration of a multi-tier system which is able to guaranty both α_{max} and ℓ_{max} performance

[3] Or more precisely, methods to build configurations, i.e. capacity planning methods.

preferences. The $OA - AL_1$ and $OA - AL_2$ configurations with an optimized architectural configuration and an ad-hoc local configuration are also able to guaranty latency and abandon rate preferences as shown in Figures 10 and 11; but this is obtained at the expense of multi-tier system cost as we will see below.

Besides performance comparison, we also evaluate and compare the cost (i.e. number of resources) of all the configurations as described in Figure 12. Obviously, $AA - AL_1$, $AA - AL_2$, $AA - OL_1$ and $AA - OL_2$ have a constant cost since their architectural configuration is fixed in an ad-hoc way, with a cost of 3 for $AA - AL_1$ and $AA - OL_1$, and a cost of 21 for $AA - AL_2$ and $AA - OL_2$. Whereas the cost of $OA - AL_1$, $OA - AL_2$ and $OA - OL$ grows when the workload increases, since their architectural configuration is optimized accordingly. $OA - AL_1$ has a higher cost than $OA - AL_2$. This is due to the fact that $OA - AL_1$ has a lower ad-hoc local configuration than $OA - AL_2$ and thus, a lower client concurrency on servers is allowed with $OA - AL_1$. Thus, in order for $OA - AL_1$ to guaranty the α_{max} abandon rate performance preference, it needs to acquire more resources in order to increase its global concurrency. $OA - OL$ presents a lower cost than $OA - AL_1$ and $OA - AL_2$ since, unlike the latters, $OA - OL$ is able to optimize the local configuration of the multi-tier system and thus, to maximize the usage of servers whenever it is possible before acquiring new resources

In summary, our experiments show that compared to approaches with a single level of local configuration where up to 94% of client requests are rejected, and compared to approaches with a single level of architectural configuration where up to 15% of resources are wasted, an approach that combines both levels of configuration allows guarantying performance preference while minimizing the cost of data centers.

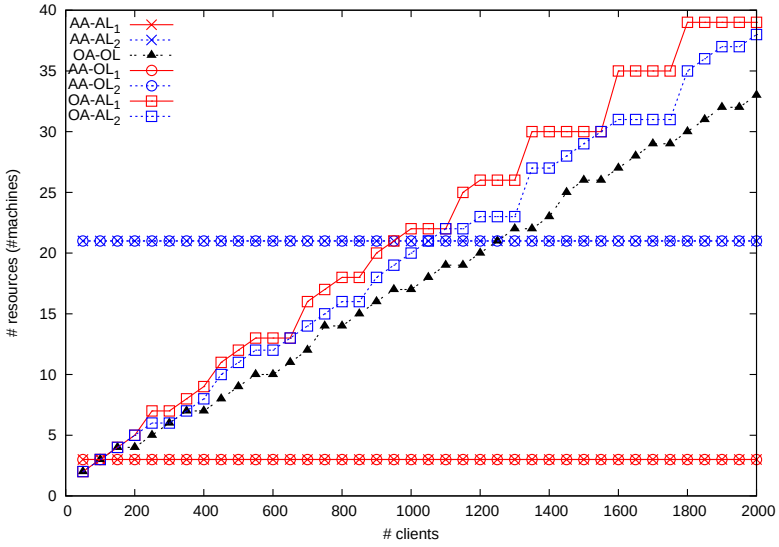


Figure 12: Local/architectural configuration – Cost

Finally, Table 4 summarizes the results of the evaluation presented in Figures 10, 11 and 12. This table provides, for each configuration, the average value

of the objective function for all considered workloads. Indeed, among all configurations that either do not consider any optimization of the multi-tier system or apply an optimization on a unique level (architectural or local), $OA - OL$ which combines the two levels of configuration provides the highest value of the objective function.

Configuration	Objective function	Accuracy (%)
$AA - AL_1$	0.04	22
$AA - AL_2$	0.04	22
$AA - OL_1$	0.04	22
$AA - OL_2$	0.05	28
$OA - AL_1$	0.15	83
$OA - AL_2$	0.17	94
$OA - OL$	0.18	100

Table 4: Auction site – Objective function

World Cup web site. To validate our approach, we also used the logs of the '98 Soccer World Cup website. We runned a 600 hour interval of these logs. Load evolution during this period of time is represented on figure 13. For clarity reasons, there is one point each 12 hours, representing the average workload amount during this period. Like with the auction site, we compare in the following 7 system configurations ($AA - AL_1$, $AA - AL_2$, $AA - OL_1$, $AA - OL_2$, $OA - AL_1$, $OA - AL_2$, $OA - OL$) of which characteristics are summarized in table 3.

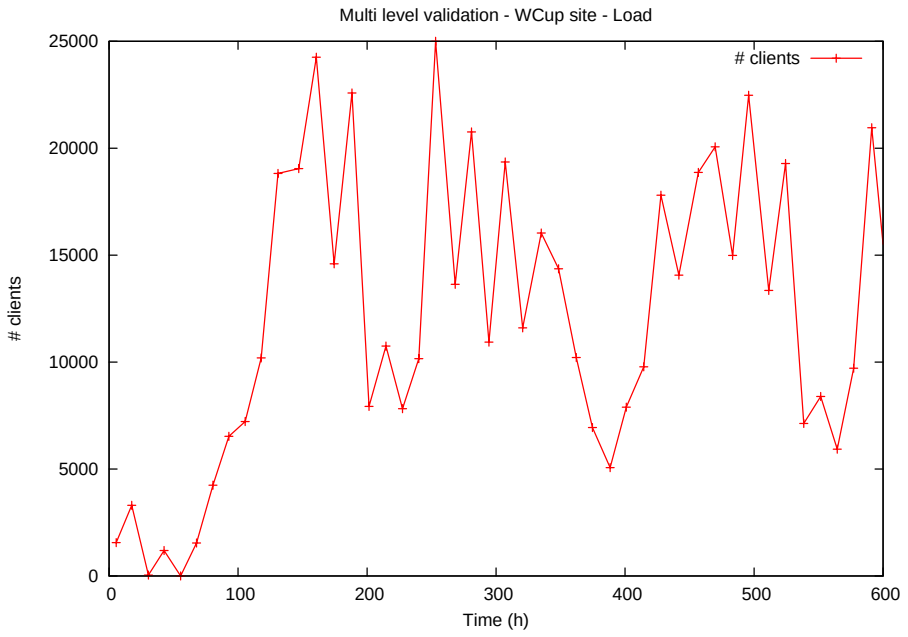


Figure 13: WCup site – Load

As we can see on figure 14, $AA - AL_2$ is the only system configuration to exceed maximum latency value. This is due to the local configuration of

300, which admits too much client to respect latency constraints. With a local configuration of 150, $AA-AL_1$ limit the amount of clients entering the system, and thus limit the overall latency. Nevertheless, lowering the local configuration without adding more resources leads to increase the abandon rate, as we can see on figure 15. The local optimization done in $AA-OL_1$ reduces abandon rate, while maintaining latency under the limit.

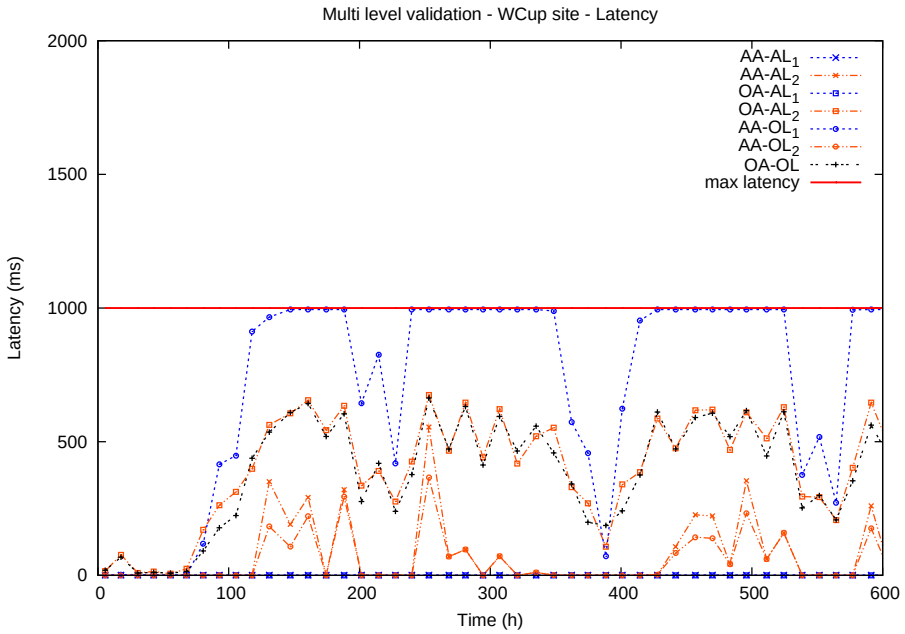


Figure 14: WCup site – Latency

However, when the amount of client grows, local optimization is insufficient and resource allocation became unavoidable. In order to match both QoS constraints (maximum latency and abandon rate), we have to perform a local and architectural optimization. Performing both local and architectural optimization improves resource consumption, as we can see on figure 16. $OA-AL_1$, $OA-AL_2$ and $OA-OL$ are all matching QoS requirement, but is the best configuration, since it uses less resources $OA-OL$ than the others.

6.5 Accuracy of capacity planning

Auction site. In the following, we will validate the accuracy of our algorithm, i.e. the ratio between the configuration returned by the exhaustive search algorithm and our algorithm.

Figure 17 compares the accuracy of the different capacity planning methods given in Table 3, namely the ratio between, on the one hand, the utility function value of the configuration returned by a capacity planning method, and on the other hand, the utility function value of the optimal configuration. The

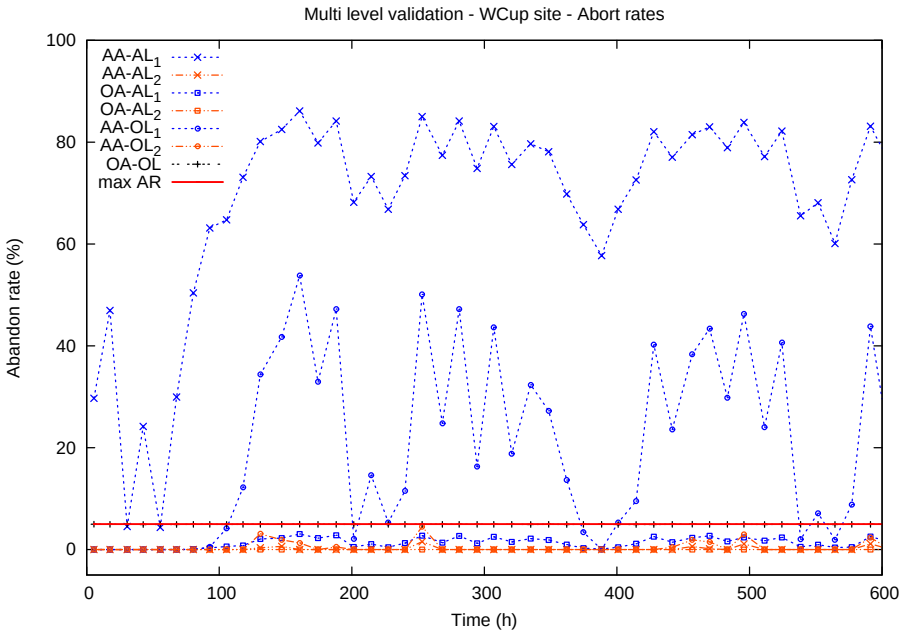


Figure 15: WCup site – Abandon rate

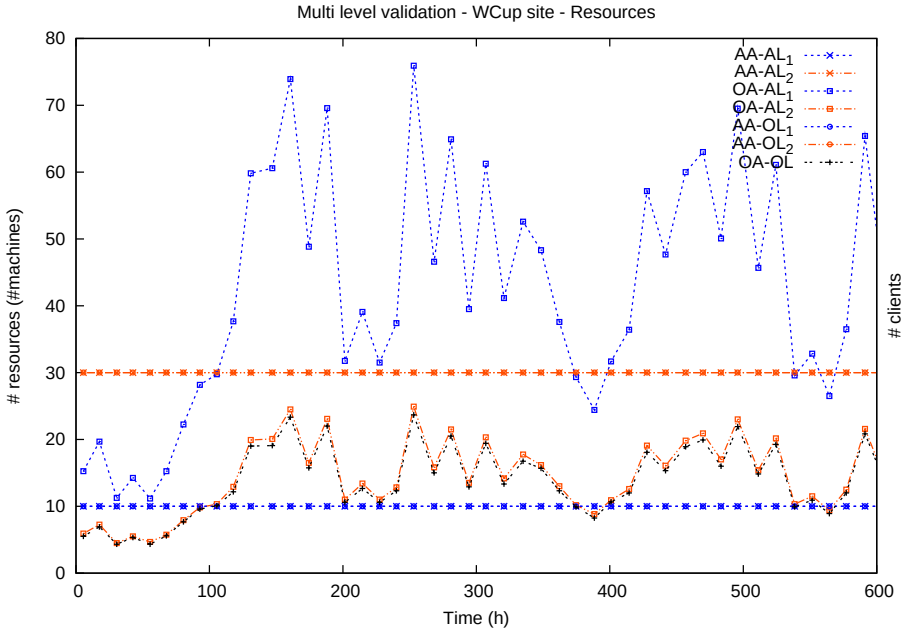


Figure 16: WCup site – Cost

numbers of Figure 17 are mean values ensued from the experiments described in Figures 10, 11 and 12.

These numbers show that our method that combines local and architectural configuration provides a 100% accurate utility with an optimal configuration of the multi-tier application.

Whereas in these experiments, a single level of architectural configuration limits accuracy between 83% and 90%, a single level of local configuration limits its accuracy between 22% and 28%, and an ad-hoc configuration accuracy is measured to 22%.

However, it is important to notice that capacity planning methods based on single local configuration or on ad-hoc approaches may induce a 0% accuracy in case of a high workload and a very low authorized maximum request abandon rate.

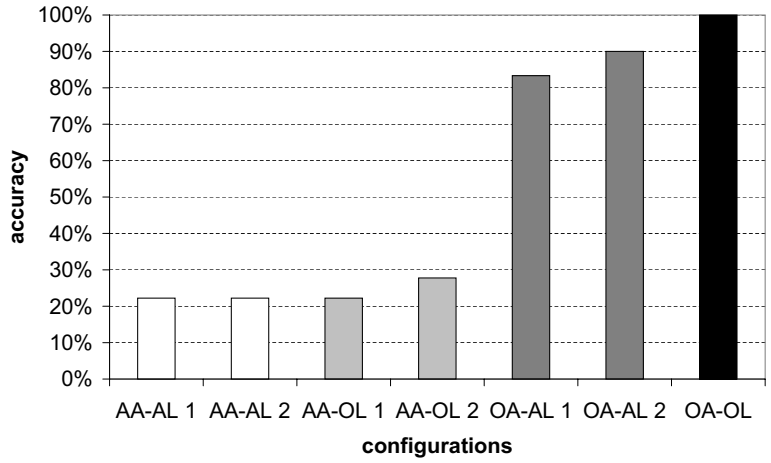


Figure 17: Accuracy of capacity planning

World Cup web site. The accuracy of our algorithm is also 100% for the World Cup Web site, as we can see on figure 18. These results are conform to the proof we made in section 4.3.

6.6 Efficiency of capacity planning

Auction site. Finally, Figure 20 compares the average processing time of the different capacity planning methods; these results ensued from the experiments presented in Figures 10, 11 and 12. Overall, the results obviously show that the more levels are considered by capacity planning methods, the more time is necessary for calculating the configuration. Nevertheless, it is important to

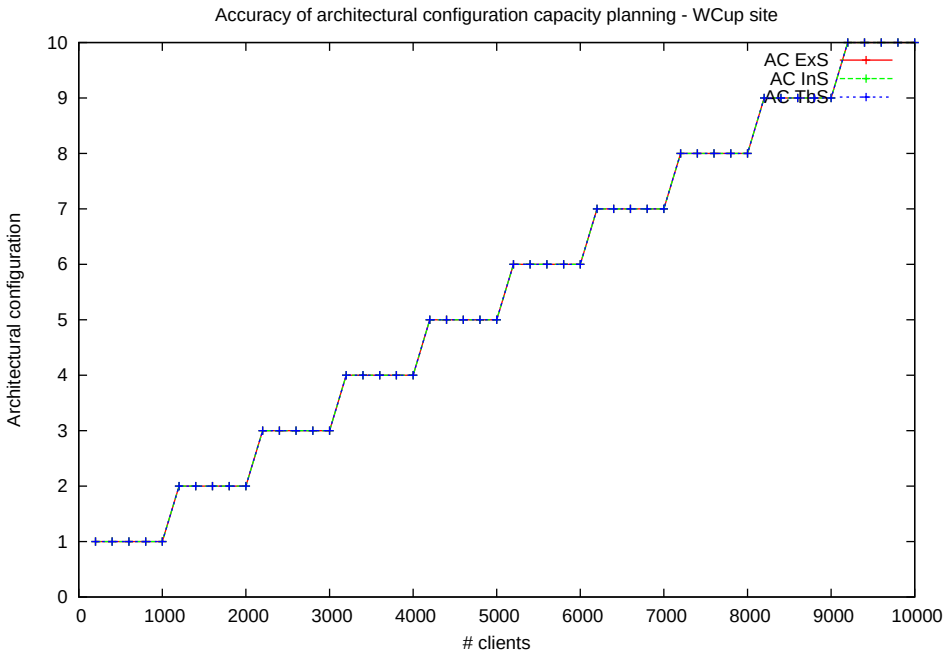


Figure 18: Accuracy of architectural capacity planning - WCup site

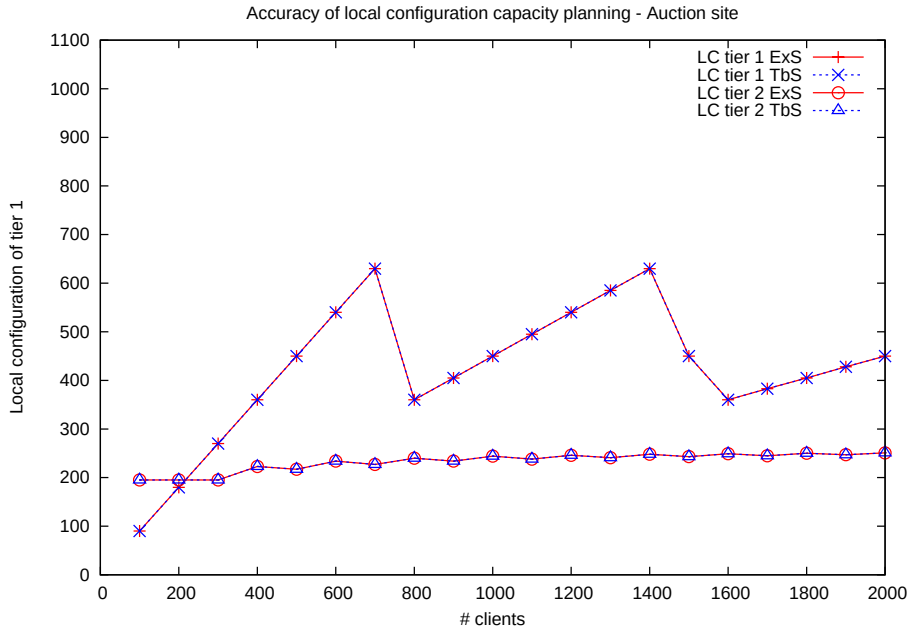


Figure 19: Accuracy of local capacity planning - Auction site

notice that compared to methods restricting their capacity planning to architectural level, a method combining local and architectural configuration does not induce an overhead. On the contrary, the latter method slightly improves the efficiency of the capacity planning algorithm since it produces smaller architectural configurations and thus, reduces the steps of the algorithm. Furthermore, for comparison purposes, we consider the case of the exhaustive search-based capacity planning algorithm (which produces an optimal local and architectural configuration). This algorithm takes 11 minutes to calculate the optimal configuration of the cluster-based two-tier auction site with a workload of 200 clients, and more than 2 hours for a workload of 600 clients.

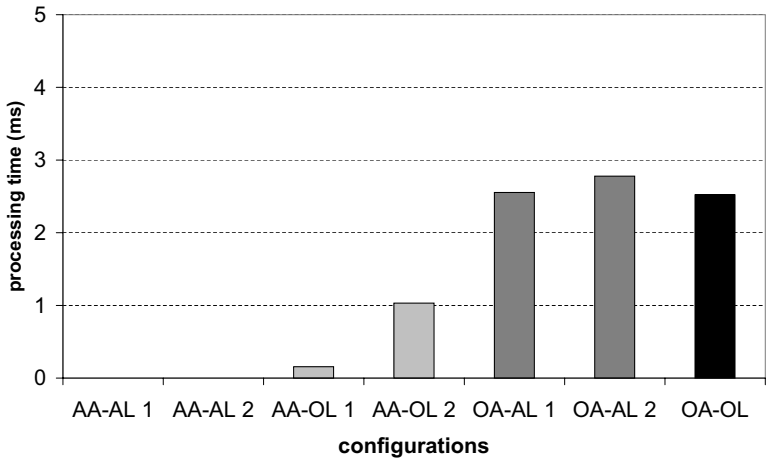


Figure 20: Efficiency of capacity planning

World Cup web site. Figure 21 present the processing times required to find the optimal configuration of the world cup web site. The tree-based search is always more efficient than the exhaustive search by several orders of magnitude.

7 Related Work

Capacity planning is a critical issue for the availability and quality of service of data centers [12]. While most of existing projects apply their techniques on a single tier [13, 20, 31, 16, 30, 14], our work differs from these projects in the fact that it tackles multi-tier data centers.

Moreover, the techniques applied to perform capacity planning may differ. On the one hand, some projects consider applying capacity planning at local level, for instance, using admission control techniques [13, 20, 31, 16, 22, 14].

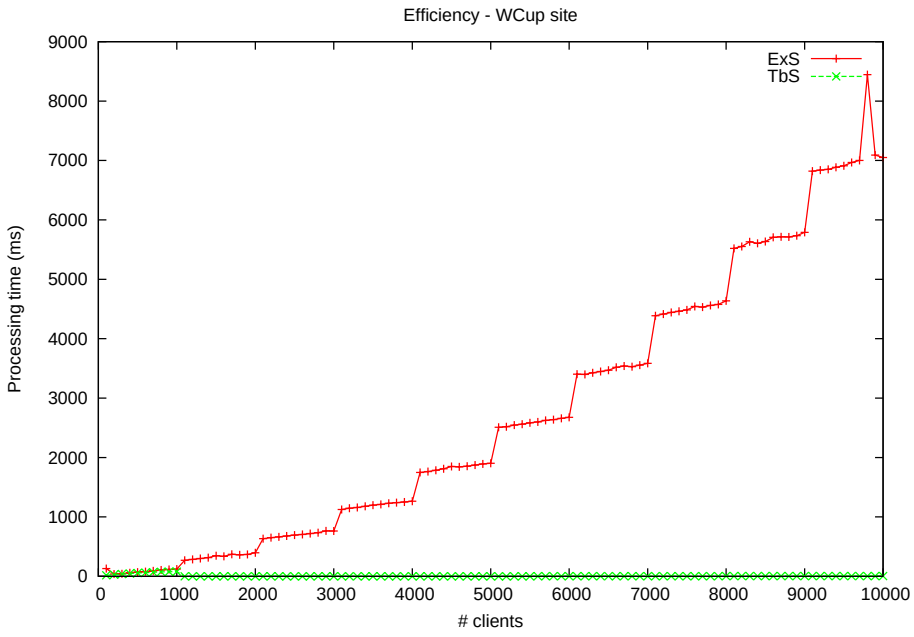


Figure 21: Efficiency of capacity planning - WCup web site

On the other hand, other projects apply capacity planning at architectural level using server provisioning techniques on cluster-based systems [29, 6, 30]. Our present work differs from other projects in the fact that it applies capacity planning of cluster-based multi-tier data centers at both local and architectural levels, with a significant gain on QoS guarantees and cost minimization.

Furthermore, in contrast to heuristics-based approaches for capacity planning of data centers that do not provide strict guarantees on QoS [6, 10, 14], the present work is based, on the one hand, on an analytic model for predicting system performance and, on the other hand, on an efficient capacity planning algorithm that controls the optimal configuration of the multi-tier data center with strict guarantees on QoS and resource usage.

The proposed model extends the Mean-Value Analysis (MVA) queuing network model [26]. It first integrates architectural configuration (i.e. server clustering) and local configuration (i.e. admission control) to multi-tier systems. It then calculates the impact of local and architectural configurations on the performance and cost of the system. Moreover, the extended model calculates new performance and cost metrics, such as client request abandon rate (one of SLA's objectives) and the cost of the cluster-based multi-tier data center in terms of numbers of machines running the data center.

8 Conclusion

In this paper, we present the design and implementation a method for utility-aware capacity planning of cluster-based multi-tier data centers. The proposed

method includes four novel features: (i) The combination of two levels of configuration of cluster-based multi-tier data centers namely architectural configuration and local configuration, this saves up to 15% of resources in a data center with a 100% accurate utility for the data center; (ii) An analytic model for cluster-based multi-tier applications that calculates application performance and cost; (iii) A utility function that characterizes the impact of local/architectural configuration of cluster-based multi-tier data centers on application performance and cost; (iv) A utility-aware capacity planning algorithm for efficiently calculating optimal configuration of multi-tier applications. These features are implemented in the MoKa modeling and capacity planning prototype. Our experiments on a cluster-based multi-tier auction site show the effectiveness of the approach.

Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see https://www.grid5000.fr).

References

[1] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and Implementation of Dynamic Web Site Benchmarks. In *The IEEE 5th Annual Workshop on Workload Characterization (WWC(5)*, Austin, TX, Nov. 2002.

[2] Apache. Apache HTTP Server. http://httpd.apache.org/.

[3] Apple. QuickTime Broadcaster. http://www.apple.com/quicktime/broadcaster/.

[4] T. I. T. Archive. Worldcup98, 2008.

[5] M. Arlitt and T. Jin. Workload characterization of the 1998 world cup web site. Technical Report HPL-1999-35R1, HP Labs, Sept. 1999.

[6] S. Bouchenak, N. D. Palma, D. Hagimont, and C. Taton. Autonomic Management of Clustered Applications. In *IEEE International Conference on Cluster Computing (Cluster 2006)*, Barcelona, Spain, Sept. 2006.

[7] B. Burke and S. Labourey. Clustering With JBoss 3.0. Oct. 2002. http://www.onjava.com/pub/a/onjava/2002/07/10/jboss.html.

[8] R. Buyya. *High Performance Cluster Computing - Volume 1*. Prentice Hall, 1999.

[9] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel. Performance Comparison of Middleware Architectures for Generating Dynamic Web Content. In *ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, June 2003.

[10] J. Chen, G. Soundararajan, and C. Amza. Autonomic provisioning of backend databases in dynamic content web servers. In *The 3rd IEEE International Conference on Autonomic Computing (ICAC 2006)*, Dublin, Ireland, June 2006.

[11] Continuent. Sequoia. http://sequoia.continuent.org/.

[12] D. A. Menascé and V. A. F. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall, 2001.

[13] Y. Diao, J. L. Hellerstein, S. Parekh, H. Shaikh, and M. Surendra. Controlling Quality of Service in Multi-Tier Web Applications. In *26th International Conference on Distributed Computing Systems (ICDCS 2006)*, Lisbon, Portugal, July 2006.

[14] S. Elnikety, J. Tracey, E. Nahum, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *The 13th international conference on World Wide Web*, 2004.

[15] X. He and O. Yang. Performance Evaluation of Distributed Web Servers under Commercial Workload. In *Embedded Internet Conference 2000*, San Jose, CA, Sept. 2000.

[16] H.-U. Heiss and R. Wagner. Adaptive load control in transaction processing systems. In *VLDB*, 1991.

[17] IBM. WebSphere Server. http://www.ibm.com/.

[18] Iron Mountain. The Business Case for Disaster Recovery Planning: Calculating the Cost of Downtime, 2001. Iron Mountain.

[19] J. Lee and R. Ben-Natan. *Integrating Service Level Agreements*. Wiley, 2002.

[20] D. A. Menasc, D. Barbara, and R. Dodge. Preserving QoS of E-Commerce Sites Through Self-Tuning: A Performance Model Approach. In *ACM Conference on Electronic Commerce (EC'01)*, Tampa, FL, Oct. 2001.

[21] Microsoft. Microsoft Exchange Server. http://www.microsoft.com/exchange/.

[22] J. Milan-Franco, R. Jimnez-Peris, M. Patino-Martinez, and B. Kemme. Adaptive middleware for data replication. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 175–194, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[23] M. Patino-Martinez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Middle-r: Consistent database replication at the middleware level. *ACM Transactions on Computer Systems (TOCS)*, 23(4), 2005.

[24] PLB. PLB - A free high-performance load balancer for Unix. http://plb.sunsite.dk/.

[25] M. Rabinovich and O. Spatscheck. *Web Caching and Replication*. 2001.

[26] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *J. ACM*, 27(2), 1980.

[27] Sun Microsystems. MySQL. http://www.mysql.com/.

[28] The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/.

[29] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. Analytic modeling of multitier internet applications. *ACM Transactions on the Web (ACM TWEB)*, 1(1):2, 2007.

[30] D. Villela, P. Pradhan, and D. Rubenstein. Provisioning servers in the application tier for e-commerce systems. *ACM Trans. Interet Technol.*, 7(1):7, 2007.

[31] Q. Zhang, L. Cherkasova, and N. Mi. A Regression-Based Analytic Model for Capacity Planning of Multi-Tier Applications. *Journal of Cluster Computing*, 11(3), 2008.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399