CloudOpt: Incremental Optimal Deployment for Dynamic Management of Clouds

Jim (Zhanwen) Li, Murray Woodside, John Chinneck, Marin Litoiu

***Abstract*— Management of large service centers and clouds requires adaptation to changing conditions and workload, which is usually based on ad hoc strategies. To obtain deployments that are close to the best possible, this paper applies large-scale optimization techniques to minimize energy use or other costs. Constraints force the satisfaction of service-level agreements with many different classes of users, and other constraints such as host memory limits. The *CloudOpt* approach determines the number of task replicas needed for application scale-out and allocates task replica execution to hosts. Determining deployments for groups of applications was earlier shown to give cost advantages over separate deployment, due to resource sharing.**

To respond to varying workloads (such as load spikes) and application addition and removal, *CloudOpt* updates the allocations periodically. Persistence techniques are used to force the deployment changes in each update to be small. The approach combines large-scale linear optimization with a non-linear performance model. It is fast enough to provide the combined deployment of several applications that combine in total about 50 different deployable units, on 500 to 1000 host nodes. Larger numbers of applications can be analyzed in groups of about this size, providing essentially unlimited scalability of the algorithm.

I. Introduction

Deployment of long-running applications (such as web applications) in a cloud must provide enough capacity to meet Quality of Service (QoS) constraints, fit the application tasks into the memory of host nodes, share hosts between applications, and minimize costs such as power consumption. The *static problem* of finding an optimal deployment of perhaps thousands of tasks on thousands of nodes is challenging; the *dynamic problem* of evolving the deployment as workload and environment conditions change is harder. This research explores the use of large-scale optimization methods and analytic performance models to find robust and agile solutions.

The scope of the system considered here is defined by the metamodel in Figure 1. It is restricted to a single datacenter (a sufficient challenge at this stage) deploying many applications, each of which may include several separately deployable units we will call "tasks" which offer services. Services in this work mean operations by the application. Replicas of each task are to be deployed to hosts so as to:

- minimize running cost, for example (but not restricted to) power consumption,
- guarantee QoS for many heterogeneous classes of users, as predicted by a performance model.
- satisfy constraints on memory and replication
- respond to dynamic changes in workload, applications, or host status, while limiting the impact on the running applications.

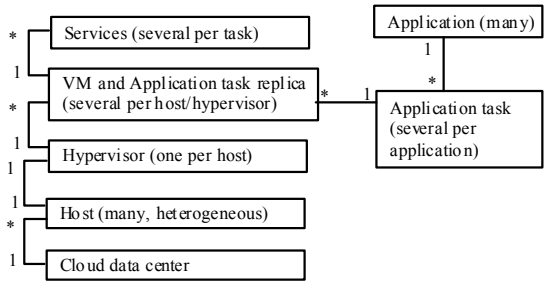


Figure 1 Metamodel of Applications to be Deployed

The overall deployment management system envisaged here is described by the larger metamodel in Figure 2. Applications are shown in the top left part, with monitoring of operational data to their right. A performance model of all the applications (middle part) is tracked and used by the optimizer (bottom part) to drive deployment decisions by the Management Platform. The optimizer (indicated by the bold dashed line) is the subject of this paper.

Most methods for deployment optimization (discussed below) do not address QoS effects directly, but constrain some proxy measure such as processor utilization to avoid excessive host congestion. As a result they cannot guarantee QoS contracts. Many of them also do not exploit sharing of resources between applications.

The CloudOpt approach (e.g. [21]) does share resources between applications, and determines optimal replication of tasks (scale-out). It simultaneously optimizes the deployment of many applications with many classes of users, including how many replicas of each task within the application should be deployed. It applies large-scale optimization techniques to obtain assurances of optimality (really near-optimality, given the approximations used); in particular it applies Mixed

Manuscript received March 5, 2012. This work was supported by the IBM Centre for Advanced Studies and by the Discovery Grant program of the Natural Sciences and Engineering Research Council of Canada.

Jim Li was with Systems and Computer Engineering, Carleton University, Ottawa Canada. He is now with the National ICT Australia. (e-mail: jimzhanwen.li@nicta.com.au)

Murray Woodside is with Systems and Computer Engineering, Carleton University, Ottawa, Canada (cmw@sce.carleton.ca)

John Chinneck is with Systems and Computer Engineering, Carleton University, Ottawa, Canada (chinneck@sce.carleton.ca)

Marin Litiou is with the School of Information Technology and the Graduate Program in the Computer Science and Engineering Department, York University, Toronto, Canada (mlitoiu@yorku.ca)

Integer Programming (MIP) and bin-packing to solve a network flow model (NFM), which is modified by QoS estimates from a dynamically tracked performance model, to enforce the QoS constraint.

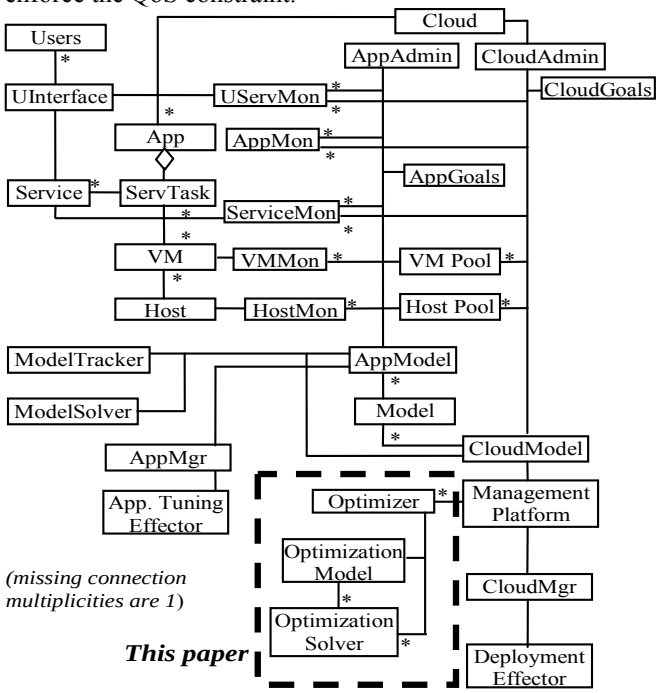


Figure 2 Cloud Deployment Management Metamodel

Dynamic optimization responds to changing workloads (such as flash crowds), new applications, component failures, and host failures and repairs. A global re-optimization after every change is too costly and may lead to thrashing of the solution. An incremental adaptive approach can:

- concentrate effort on essential changes, e.g. to meet QoS constraints, to repair failures and to deploy new applications.
- avoid low-value shifts in deployment, with attendant costs and delays for reconfiguration,
- reduce disturbance to the running applications,

It should balance optimality against a preference to retain the existing deployments. The optimization literature calls this *optimization with persistence* [6]. This paper evaluates persistence strategies for deployment optimization, to see if they are effective in reducing the changes in deployment while staying close to the optimal cost. The purpose is to establish the applicability of the combination of large-scale formal optimization methods with persistence, to a problem dominated by ad-hoc adaptive methods.

To deal with the dynamic situation and control the perturbation to the running system, persistence mechanisms modify the optimization problem by including:

- penalties for creation of new replicas (including moving a replica to a different host),
- penalties for the use of new hosts (hosts not used in the previous solution),
- rewards for unchanged replica/host combinations.

The contribution of the present work is to design novel practical strategies using these mechanisms, and to evaluate them, as described in Sections V and VI. Two of these three mechanisms give good results. The results show that dynamic optimization can be applied to this problem at moderately large scales, and by sub-dividing the problem they can give effective large-scale optimization of time-varying cloud deployments at any scale.

The information about the changing situation is obtainable from the system monitoring and management functions. Static and dynamic CloudOpt use information about the application and its workload and QoS from a performance model (described below) whose parameters are continuously tracked as described in [40].

To control the complexity of the study, restrictions were placed on the generality of the analysis:

- host and software component failure and repair were not modeled, although CloudOpt can handle them,
- communications delays are not considered explicitly

Communications delays were not modeled, except for the network delay between user and cloud. Homogeneous message transfer delays within the data center can be handled by the present method.

A. Related Work on Static Optimization of Cloud Deployments

The simplest optimization model considers only load balancing and processing capacity, and fitting the necessary task replicas into the set of machines that has the lowest total cost. Resource contention is ignored. *Network Flow Models* (NFMs) allocate the execution flow of applications to hosts to fit their capacities, using linear programming (e.g. [18]). Optimization is always at least as good as load balancing.

Mixed Integer Programming (MIP) extends linear programming with integer constraints due (for example) to limited memory space, e.g. [8], [32], and (combined with heuristics) [31]. MIP has also been used to optimize power [2] [26]. However MIP does not scale well because of the inherent tree-structured search in all solution methods, and also does not address QoS directly. *Knapsack optimization* methods as used in [3] [13] have characteristics similar to MIP.

Bin-packing heuristics (e.g. [7], [10], [30], [31]) are much faster than the methods listed above and scale well, but produce worse values of the objective function. Bin-packing becomes complex when there are multiple kinds of constraints ("multidimensional bin-packing" [9]). In [27] two layers of packing heuristics were used to find optimal deployments with multiple objectives, for Hadoop-MapReduce systems.

None of these methods consider resource contention, which makes the objective function nonlinear. However a *utility function* approach [34] can combine performance predictions with penalties for other objectives into a single nonlinear objective function by weighting. Examples include:

- *Beam Search,* used in [24] to minimize a general nonlinear utility function which includes QoS estimates from a performance model.
- *Hill climbing* for maximizing system workloads in service systems [23], minimizing replicas to provide a required QoS

[39], seeking optimal service components in distributed systems [25], and allocating resources for DBMS [38]. Hill-climbing is limited by the time and cost to do many evaluations of utility.

The issue of contention delays has also been addressed in [37] by applying nonlinear integer optimization to the results of a queuing model to determine the number of hosts required by a multi-tier server network subject to QoS requirements. That work does not however consider other issues such as memory, and the location of replicas.

Some studies consider only communications costs when deploying tasks, for example *graph algorithms* such as min-cut (e.g. [4]). They are efficient, but work best on just two nodes or two subnets, so they are difficult to scale up for large systems. Bin-packing was adapted for communications cost in the Multifit-Com algorithm [35] (which considers capacity limits on both communications and processing). Communications loading can be included in MIP models and in heuristics such as the one described in [31].

Static CloudOpt [17][18][19][21] deals with nonlinear performance predictions by combining a linear network flow model (NFM) solution with a performance model, iterating between the two. The method outlined in [18] is extended in [17] to accommodate one new application at a time. In [19] bin-packing alone is applied to meet constraints on host memory and on software licenses. In [21] bin-packing and MIP are combined to improve scalability.

B. Related Work on Dynamic Optimization of Cloud Deployments

Dynamic optimization adjusts the deployment as the system workload changes over time. One simple approach is to perform a new static optimization either periodically or in response to a change. This was done in [24], which applied *Beam Search* to maximize a utility function of response times found by a performance model. Its scalability is in principle good (polynomial complexity in problem size) but its effectiveness depends critically on heuristic criteria which may be difficult to define. The method was effective in experiments on 2 applications with multiple classes of users, on 30 identical hosts.

Feedback control makes decisions as functions of some performance measures (such as processor utilization). Ad hoc change rules triggered by threshold levels of measures are common in practice, e.g. adding a process replica when utilizations pass a threshold, or deleting it when they drop below a (lower) threshold (e.g. [5]). Ad hoc rules can be successful, but they are difficult to choose and to manage, and they interact in unpredictable ways (typically they ignore interactions between their decisions).

Single deployment moves controlled by heuristic rules can be effective in updating a system after a small change in conditions (they are a feature of e.g. [31]). However they lack a solid basis and cannot always cope with situations requiring multiple moves to obtain an improvement.

Reinforcement learning (RL) can be applied to small scale dynamic systems. It has been used to seek optimal resource allocation decisions under changing workloads [33], to optimize configurations in virtual environments [27] and to configure database workloads on virtual machines [29].

Network flow models (NFMs) were used in [13] to optimally schedule the next steps in large batch computations (such as MapReduce) in which data is stored in the nodes, and the next steps must communicate to obtain it. This use of NFMs (like ours) can also solve large problems, but it is more fine-grained in time and does not consider long-running tasks.

In [29] Tang et al. employed a NFM similar to the NFM in CloudOpt, but only to maximize the flows within a given deployment. Deployment is adapted by a method based on single moves, and accommodates as much offered load as possible within the system's capacity. The algorithm has (according to [29]) been implemented commercially (in Websphere) and results are given for up to 17,500 applications running on 7000 hosts. However, this method does not attempt to guarantee QoS, and handles only single-task applications.

This paper extends [21] for a dynamically changing environment by applying persistence techniques to modify the optimization problem. It balances all deployment decisions simultaneously, subject to guidance (in the form of constraints or costs) that limit the total amount of change. The results show that two persistence approaches are effective, insensitive to several variations in the problem, and scale well to at least moderate sized systems. Further scaling can benefit from partitioning the problem.

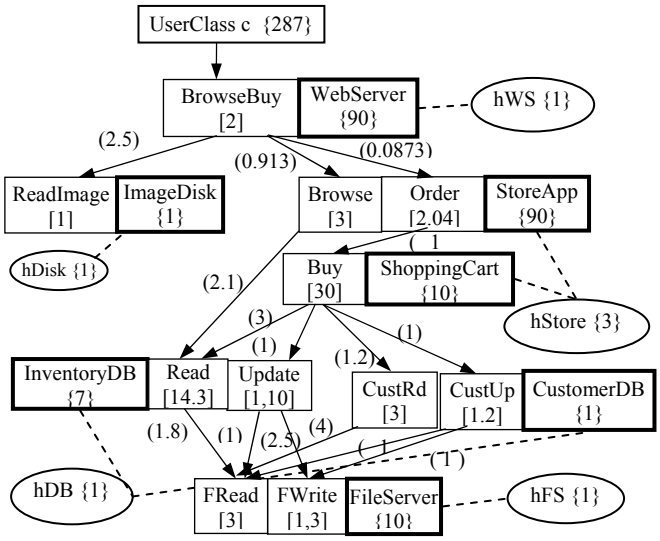


Figure 3 An example of an Application Template Model, showing a layered application architecture and illustrating the LQN notation, as annotations on the architecture.

II. MULTIPLE APPLICATIONS IN A CLOUD

We consider service-oriented applications (for instance enterprise, social networking, and web service applications) with a layered service architecture as illustrated by an example in Figure 3. Groups of *users* access *services* (shown by non-bold rectangles) offered by deployable multi-threaded *service tasks* (the attached bold rectangles). Services use lower layer services through request-reply interactions indicated by call arcs (arrows) between the services. Each task with its VM is deployed on a host shown as an associated ellipse. Task *t* has a multiplicity $\{m_t\}$ (for its thread pool size) and host *h* has a

multiplicity $\{m_h\}$ (for multi-core or multiprocessors).

Each application has one or more classes of users with N_c users, a thinking time between responses of Z_c sec, and a mean response time R_c sec for class c. The desired QoS guarantee requires that R_c be less than a specified value R^*_c:

$$R_c \le R^*_c,\ c = 1,..., C \quad (1)$$

If the QoS requirements of class-c users are expressed as a percentile of responses (e.g. 95% less than 0.5 sec), then a corresponding requirement on R_c can be found from the ratio of the average to the percentile, which can be estimated online from the shape of the measured distribution.

A. Performance Estimation for the Application

Performance-related parameters are attached to the Figure, and are used by the performance model. The execution demand of service s is $[d_s]$ (e.g. the label [3] on Browse means 3 time units of CPU execution for the task and its VM and hypervisor, per invocation of the service). The mean requests to other services is (y) annotated on a call arc (thus, each operation by StoreAccess makes 0.913 requests to Browse). These parameters are estimated by the least-squares tracking estimator described in [40] and indicated in Figure 1.

Host demands are calibrated on a chosen "standard" processor type. Host h has a speed factor σ_h relative to this, and the actual CPU demand is d_s/σ_h.

The layered queueing network (LQN) performance model uses the information shown in Figure 3 (in LQN, the services are called entries). LQN is an extended queueing model which solves for contention for processing and thread resources, and estimates the response times and throughputs of services. LQN modeling techniques are described in [11][28][36].

The optimization uses the corresponding constraint on the class throughput f_c, which is obtained from R^*_c and the class parameters N_c and Z_c using Little's identity $f_c = N_c/(R_c + Z_c)$ responses/s. Thus

$$f_c \ge f^*_c = N_c/(R^*_c + Z_c) \quad (2)$$

It is convenient in LQN to define a user population N_c, but if it is unknown, it can be set to any value much larger than the expected number of active users, and Z_c set to zero.

Figure 1 shows each task without replication, and with a dummy deployment; this will be called an *application template model*. When the replicas are defined with all their deployments, a task copy is created for each replica. This gives the *deployed LQN* performance model, which is evaluated for that scaling and deployment.

III. Optimization Model

A. Decision Variables (Capacity Allocation)

The optimization determines a deployment matrix $\boldsymbol{\alpha}$ giving the capacity allocation α_{ht} of task t to host h, for all tasks and hosts:

$$\boldsymbol{\alpha} = (\alpha_{ht}),\ \alpha_{ht} = \text{capacity reserved for task } t \text{ on host } h$$
$$t = 1,...,T;\ h = 1,...,H$$

The response times and throughput of class c users under deployment $\boldsymbol{\alpha}$ are then $R_c(\boldsymbol{\alpha})$ and $f_c(\boldsymbol{\alpha})$ respectively.

Each task is a deployable unit in its own virtual machine. It is assumed that any number of replicas of each task may be deployed, so α_{ht} may be non-zero for multiple hosts. Each host is a computing node, possibly a multiprocessor, with an available capacity Ω_h. The units of $\boldsymbol{\alpha}$ and Ω_h are units of utilization as measured on a chosen "standard processor", such that a faster processor or multicore processor has a larger capacity. From $\boldsymbol{\alpha}$ we can derive the useful matrices of indicator variables, $\boldsymbol{A}$ and $\boldsymbol{S}$:

$$A_{ht} = 1 \text{ if } \alpha_{ht} > 0, \text{ indicating task } t \text{ is deployed on host } h, \text{ else zero} \quad (3)$$

$$S_h = 1 \text{ if } \max_t(\alpha_{ht}) > 0, \text{ indicating host } h \text{ is used, else zero.} \quad (4)$$

B. Execution Cost

The goal is to minimize a cost function which includes the cost of using the hosts. For each host h there is a fixed cost $C_{F,h}$ which is incurred if h is used at all, and a variable cost $C_{V,h}\Sigma_t\alpha_{ht}$ proportional to its total allocated capacity $\Sigma_t\alpha_{ht}$. This gives a total execution cost for all hosts as:

$$ExecCost = \Sigma_h\,(C_{F,h}\,S_h + C_{V,h}\,\Sigma_t\alpha_{ht}) \quad (5)$$

The optimization will minimize a total cost which includes this sum.

Our preferred cost function is energy consumption. As in [14], the electrical power used by host h is approximated by the sum of a fixed power C_{Fh} at zero utilization, and a term proportional to the utilization $C_{V,h}U_h = C_{V,h}\,\Sigma_t\alpha_{ht}$. This gives the linear curve in Figure 4, and *ExecCost* in Eq (5) then models energy consumption.

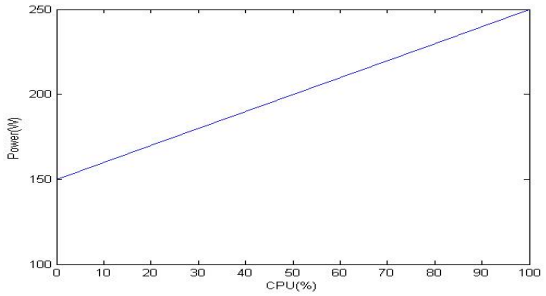


Figure 4 Linear approximation of power consumption vs. CPU utilization, based on [14]

C. Host Capacity Constraint

Capacity is allocated up to a host's "available capacity" Ω_h which is defined for host h as:

$$\Omega_h = (\text{CPU speed ratio relative to a standard processor}) \times (m_h = \text{number of CPUs or cores}) \times (\varphi_h = \text{max permitted utilization per CPU} < 1) \quad (6)$$

The optional factor φ_h artificially forces some spare physical capacity which is always needed anyway, and which helps the iterative algorithm described below to achieve feasibility more quickly; $\varphi_h = 0.8$ in the experiments below.

Then the allocations $\boldsymbol{\alpha}$ must satisfy

$$\Sigma_t\alpha_{ht} \le \Omega_h,\ h = 1, H \quad (7)$$

The units of Ω_h and α_{ht} are those of a host utilization (cpu seconds executed, per second, on a standard host).

The capacity requirements of the tasks are determined by the user throughputs and the execution demands of the tasks as estimated by the performance model parameter tracking. One class c user request generates a demand by task t for CPU time $d^*_{c,t}$ in seconds on a "standard" processor. Sufficient capacity for task t requires $\Sigma_h \alpha_{ht} \geq \Sigma_c f_c d^*_{c,t}$, but since cost increases with $\boldsymbol{\alpha}$ the equality is always optimal. Thus we set

$$\Sigma_h \alpha_{ht} = \Sigma_c f_c d^*_{c,t} \quad (8)$$

The task demand $d^*_{c,t}$ is determined from the application template LQN as

$d^*_{c,t}$ = Σ_s (number of invocations of service s during a class-c response, found from the call parameters) × (service demand d_s for the entry)

where the sum is over the services of task t.

D. SLA Constraint on Performance

The service level agreement for each class of users defines the constraint (using throughputs, considering Eq (2))

$$f_c(\boldsymbol{\alpha}) \geq f_c^* \quad (9)$$

where $f_c(\boldsymbol{\alpha})$ is computed by the performance model. The optimization model nominally satisfies the constraint but it ignores the effects of contention in the system, which reduces the throughput (by increasing the delay). Thus Eq (9) may not be satisfied when the performance is evaluated.

To force the satisfaction of Eq (9) an optimization sub-problem (a MIP) is constructed with an artificial surrogate constraint, that forces additional capacity to reduce the contention delays:

$$f_c = f_c^* + \Delta_c \quad \text{(surrogate constraint for performance)} \quad (11)$$

where Δ_c is an artificial flow component determined from the performance model solution (and initially set to zero).

E. Memory and License Constraints

To fit tasks requiring memory M'_t for each replica of task t into a memory of size M_h for host h, requires

$$\Sigma_t A_{ht} M'_t \leq M_h, \quad h = 1,H \quad (12)$$

There may also be a constraint on the number of replicas of task t, if for instance a finite number L_t of licenses are owned for task t. Instead of a hard constraint, a cost penalty for replicas beyond that number is more appropriate; it can be expressed as a cost C_L for each license beyond L_t:

$$LicenseCost = \Sigma_t C_L(\max(0, \Sigma_h A_{ht} - L_t)) \quad (13)$$

F. MIP Summary

To deploy T tasks on H hosts requires the optimal choice of α_{ht}, for all combinations of host h and task t, to minimize the total cost given by Eq (5) and (13) as:

$$Cost = ExecCost + LicenseCost \quad (14)$$

subject to constraints in Eqs (3), (7), (8), (11), and (12). This is a mixed integer programming (MIP) problem (integer because of the license and memory constraints, and the definition of A). This MIP is referred to as a sub-problem because it is constructed and solved repeatedly during an iterative solution.

G. The Performance Model for Solution $\boldsymbol{\alpha}$.

CloudOpt uses a performance model to calculate $f_c(\boldsymbol{\alpha})$, from each solution for $\boldsymbol{\alpha}$. For a deployment $\boldsymbol{\alpha}$, a *deployed LQN* is created from the architecture template. For every host with A_{ht}=1 an instance of task t is created on host h. For multiple replicas of a task t, the requests to t are divided between the replicas in proportion to the allocation α_{ht}. On host h the scheduler divides the share of the CPU in proportion to α_{ht} for each task t deployed there.

IV. The Static CloudOpt Algorithm

In CloudOpt the optimization is carried out iteratively. A solution is first found to the MIP with $f_c \geq f^*_c$, and a performance model is created for the resulting deployment. The performance model is solved, and for the first step its throughput is denoted $f^{(1)}_c$. Resource contention generally makes $R_c > R_c^*$ and thus $f^{(1)}_c < f_c^*$. The throughput shortfall is denoted $\delta^{(1)}_c$ for the first iteration:

$$\delta^{(1)}_c = f_c^* - f^{(1)}_c. \quad (15)$$

In the MIP, a surrogate user throughput flow requirement $\Delta^{(i)}_c$ is added after iteration step i for each user class c, such that $\Delta^{(i)}_c$ is the sum of the shortfalls $\delta^{(j)}_c$ at all the steps up to and including the ith:

$$\Delta^{(i)}_c = \Sigma_{j=1}^{\,i} \delta^{(j)}_c \quad (16)$$

Then at step $i+1$ the MIP is solved with $f_c = f^*_c + \Delta^{(i)}_c$.

The surrogate flow increases the capacity which must be provided for tasks, which reduces contention and increases throughput (as calculated by the performance model). CloudOpt stops at the first feasible solution (in which $f^{(j)}_c \geq f^*_c$ for all c).

To improve scalability, there is also a preliminary bin-packing step to determine a reduced candidate set of hosts and reduce the complexity of the MIP. The algorithm is summarized as follows:

Static CloudOpt Algorithm

Step 1. Construct the Application Template Models from application knowledge and measured data. Construct the optimization model (MIP with its network flow model) from the Application Template Models and the available hosts.

Step 2. Apply heuristic bin-packing to select a subset of hosts.

Step 3. Apply MIP to the reduced problem.

Step 4. Construct the deployed LQN to incorporate the replication and deployment decisions of Step 3.

Step 5. Solve the deployed LQN.

Step 6. Test the feasibility of the solution with respect to the QoS requirements:

Step 6a. If not feasible, update the surrogate flows in the optimization model, and repeat from Step 3.

Step 6b. If feasible, terminate.

While there is no guarantee of feasibility this algorithm has always given a feasible solution in a few steps provided the hosts have sufficient capacity. Iterations beyond first feasibility give slow convergence, so a final step called a Linearized End Step was designed to find a final solution.

V. Dynamic Conditions

Dynamic conditions require re-optimization, which incurs two kinds of *reconfiguration costs* for deployment changes.

- a cost for starting up a host which was previously idle, the "new host" cost C_host_h for host h,
- a cost for starting up a replica of a task, the "new replica" cost C_rep_t for task t.

For simplicity, changes to workload and the deployed applications, and their re-optimization, are assumed to occur at periodic step times. This can easily be generalized to provide re-optimization whenever a change occurs, and the evaluation should not be affected by the approximation provided the steps occur frequently enough (relative to change frequency).

Five strategies were compared, called *NoPer, P-Rule, P-Rwd, P-PenRep,* and *P-PenRH*.

Four persistence strategies were compared against the baseline "strategy" *NoPer*:

NoPer (*No Persistence*) strategy uses static CloudOpt at every step, without persistence, and is used for comparison. It should give the lowest cost.

A. Persistence Strategies

Brown et al. [5] suggest obtaining persistence by (i) using constraints to limit deviations from the previous solution, or (ii) using penalties or rewards on deviations from the previous solution.

Constraints are applied using upper and lower bounds on the flow allocations α, we apply the constraint:

$$\alpha_{ht,lower} \le \alpha_{ht} \le \alpha_{ht,upper}$$

Considering the previous allocations α_prev,

- to preserve a previous deployment, but let the allocations change, set $\alpha_{ht,upper} = 0$ for all h,t for which $\alpha\_prev_{ht} = 0$,
- to preserve a previous allocation exactly, set $\alpha_{ht,lower} = \alpha_{ht,upper} = \alpha\_prev_{ht}$,
- to disable a specific allocation set $\alpha_{ht,upper} = 0$

Penalties and Rewards: Because it is impossible to ensure that there is a feasible solution with hard constraints on flows, we prefer to apply penalties to flows that violate persistence. With a penalty, the objective function becomes

$$Total\ Cost = (Running\ Cost) + Penalty \quad (17)$$

where the running cost is given by Eq (14) and the penalty is a sum of terms for different deployment changes. Rewards are expressed by negative penalty terms.

P-Rule (*Persistence by Rule*) uses constraints and an incremental strategy. It deletes applications which terminate, and optimally allocates each new application to a separate set of idle hosts. Other applications are unchanged, except for adding replicas after an increase in workload. *P-Rule* applies static CloudOpt to new applications or those with increased workload, with two additional constraints called *P-Rule-A* and *P-Rule-*B:

P-Rule-A: The capacity allocations for previously deployed tasks are not reduced, expressed by a bound:

$$\alpha_{ht} \ge \text{previous } \alpha_{ht} \quad (18)$$

P-Rule-B: There are no new deployments on already-used hosts, expressed by a bound for new tasks:

$$\alpha_{ht} \le 0 \text{ if the previous } S_h > 0 \text{ (host was used)} \quad (19)$$

$$\alpha_{ht} \le \text{infinity otherwise}$$

In *P-Rule* the solution tends to degrade over time, which can be countered by a full optimization at every *n*th step, as in [20]. This was not included in the present study.

The other three strategies use penalties within a full optimization at every step. They use the parameters defined in Table 1 below, and are defined as follows:

P-Rwd (*Persistence by rewards*) strategy adds a reward for each task whose deployment is unchanged. A negative penalty ($-C\_unch_t$) is given for each previous deployment of a replica of task t which is retained. The penalty is

$$Penalty = -\Sigma_{ht}\, C\_unch_t\, A\_prev_{ht}\, A_{ht} \quad (20)$$

P-PenRep (*Persistence by Penalties for new Replicas*) strategy penalizes each new replica of task t above a stated fraction *frac* of its previously deployed replicas by an amount C_rep (the same penalty for all tasks), giving:

$$Penalty = C\_rep\ \max(0, (\Sigma_{ht} A_{ht} (1-A\_prev_{ht})) - frac\Sigma_{ht} A\_prev_{ht}) \quad (21)$$

P-PenRH (*Persistence by Penalties for all new Replicas and Hosts*) strategy penalizes each new replica by the penalty C_rep and each new host by the penalty C_host_h specific to the host). The number of new replicas is $\max(0, (\Sigma_{ht} A_{ht} (1-A\_prev_{ht}))$, and of new hosts is $\Sigma_h(1-S\_prev_h)\, S_h$, giving:

$$Penalty = C\_rep\ \max(0, (\Sigma_{ht} A_{ht} (1-A\_prev_{ht})) + C\_host_h \Sigma_h (1-S\_prev_h)\, S_h \quad (22)$$

The parameters of the penalty functions can be adjusted to put more or less weight on persistence versus cost optimization.

Table 1 Parameters used in the penalty strategies

A_prev_{ht}	The value of A_{ht} in the previous solution (task t was deployed on host h)
C_unch_t	The reward for a retained replica.
S_prev_h	The value of S_h in the previous solution (host h was used)
C_rep	Startup penalty for a new task replica
C_host_h	Start-up penalty to use a new host h
frac	Unpenalized new replicas in one strategy (see Eq (21))

Note that because the system performance is constrained to match the QoS specification, the QoS (the response time or throughput) is essentially identical for all the strategies.

A summary of the dynamic CloudOpt algorithm follows:

Dynamic CloudOpt Algorithm

Step 1. Choose a strategy (*NoPer*, *P-Rule*, *P-Rwd*, *P-PenRep*, *P-PenRH*) and values for the parameters *C_rep, C_host, C_unch, frac*.

Step 2. Apply static CloudOpt to the initial configuration.

Step 3. At each time step for dynamic changes:

3a. Create an Application Template Model for any new application to be deployed.

3b. Update the Application Template Model for each running application, by the ModelTracking estimator indicated in Figure 2.

3c. Create an appropriate MIP for the deployment, depending on the strategy:

- Minimize *Total Cost* in place of *Cost*, using *Penalty* as defined for the strategy,
- For *P-Rule* add the constraints in Eq (18) and (19).

3d. Apply Steps 2 - 6 of Static CloudOpt to find the new deployment.

VI. Experiments and Evaluation

A. Experimental System and Conditions

The five strategies of Section V were executed on trace data. The applications all have the architecture of Figure 3 with seven independent tasks and ten services, and with different randomly generated parameters (for CPU demand for services, memory requirement, and license availability).

The applications are to be deployed on a host pool with five types of hosts, with the speed and cost parameters shown in Table 2. The costs C_{Vh} and C_{Fh} have arbitrarily chosen units, but the ratios between C_{Vh} and C_{Fh} are based approximately on the experimental results in [22]. Host type A is taken as the "standard host" used to evaluate the service demands, and the others have speed ratios relative to host type A. Each type of host in the pool can host any task. In each experiment there are roughly equal numbers of hosts of each type.

Table 2 Host data

Host Type	Speed Ratio	Execution Cost (C_{Vh})	Fixed Cost (C_{Fh})
A	1.00	0.32	1.61
B	1.33	0.27	1.62
C	1.67	0.222	1.63
D	2.00	0.215	1.64
E	2.67	0.21375	1.66

The size of the host pool determines how tightly the applications fit into the available processing resources. A situation with few excess resources is a "high-stress" situation, and one with plenty of resources, low stress. The demand parameters were multiplied by a suitable factor to set the *stress rate* of the workload defined as:

$$\textit{Stress Rate} = (\text{total demand})/(\text{total host capacity}) \qquad (23)$$

The strategy parameters in Table 1 were chosen as follows:

- $C\_unch_{ht} = \alpha_{ht}$ was chosen to reward existing deployments in proportion to the loads on each replica. This tends to preserve replicas with larger host utilizations. Thus for *P-Rwd*, Eq (20) becomes

$$\textit{Penalty} = -\Sigma_{ht}\alpha_{ht}\, A\_prev_{ht}\, A_{ht} \qquad (24)$$

- *C_host* = 5 units of start-up penalty for a new host to boot up the computer (used in Eq (22))
- *C_rep* = 3 units of start-up penalty for a new replica to install the VM and image (used in Eqs (21), (22)).
- *frac*=20%
- For *P-PenRep*, the value of *C_rep* is replaced by a large number (about $4x10^9$) to forbid exceeding the target,
- the stress rate was varied as described below.

The evaluation was performed on four traces of varying workloads, three (called WorkVar cases) describing varying intensity of requests and users for a single application, and one (called the AppVar case) describing a varying number of different applications. The evaluation attempts to answer the following five questions.

Questions for the Evaluation

1. Is persistence effective, in the sense that new hosts and new replicas are greatly reduced relative to *NoPer*? And, which is the best strategy among the ones tested?
2. Is the running cost much higher using persistence, relative to *NoPer*? Also, does the running cost under persistence deteriorate over time, relative to *NoPer*?
3. How does the amount of change in the workload, over one step, affect the performance of the strategies?
4. How does the relative weight given to reconfiguration changes in *Total Cost* affect the performance of the strategies?
5. How does the execution time of the strategies compare to *NoPer*, and how is it affected by the size of the problem?

B. Results for Varying Workloads (WorkVar Cases)

The first group of experiments has a single application with a varying workload intensity. Three traffic intensity traces were used, based on recorded workload traces:

- WorkVar_soccer: based on 1998 world cup soccer trace [1],
- WorkVar_web: based on a web traffic trace described by Akamai Technologies [16],
- WorkVar_media: based on a multimedia workload traffic trace from [16].

The recorded traces were re-expressed as a varying number of users with a think time of 950ms to give a trace of the equivalent number of users with 72 time-steps of 20 minutes each, shown in Figure 2. The traces from [16] were derived from graphs published for a keynote presentation.

The size of the host pool was fixed with an initial stress rate of 0.1, in which the pool includes three of host types A and B, and four of host types D and E as defined in Table 2.

1) Result Traces

The step-by-step results for running costs and new replicas are shown for the WorkVar_soccer trace in Figures 5 and 6. The comparison of the five strategies is similar for results from the other traces.

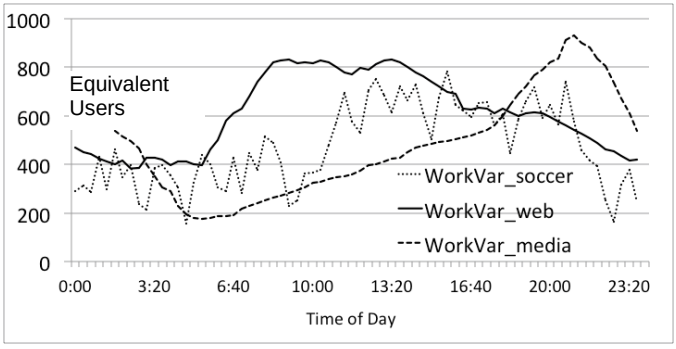


Figure 5 Three workload intensity patterns (WorkVar traces)

From lowest to highest running-cost the five strategies are roughly in the order: *NoPer, P-PenRep, P-Rwd, P-PenRH, P-Rule.* The first and last of these are expected, since *NoPer* minimizes only running cost, while *P-Rule* mostly ignores resource sharing between applications. Sometimes *P-PenRH* gives equal or lower cost than *P-Rwd*, and sometimes *P-Rwd*

and *P-PenRep* are both as good as *NoPer*.

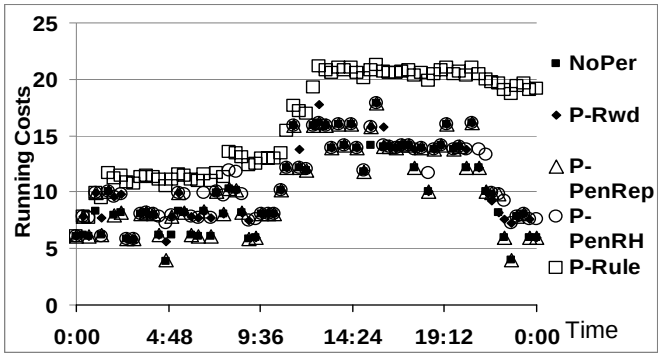


Figure 6 Running cost for the WorkVar_soccer trace

The new replica results in Figure 7 put the strategies in roughly the opposite order, with the least new replicas for *P-Rule* and then *P-PenRH, P-PenRep, P-Rwd, NoPer*. Appendix VI-2 gives the results for the other WorkVar cases, and for new hosts in all cases, which also support this observation.

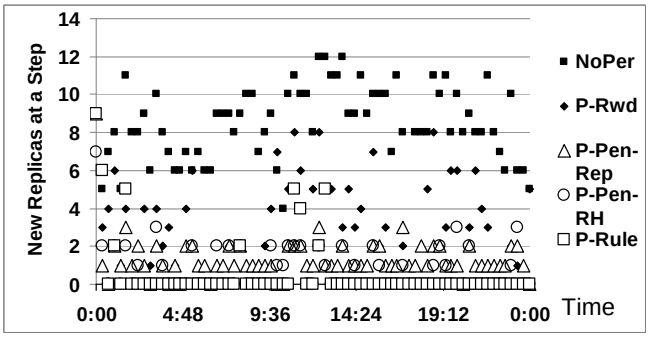


Figure 7 New replicas for WorkVar_soccer

A statistical summary shows the differences more clearly. Figure 8 (a) - (c) shows box plots giving the median, quartiles, and maximum and minimum of the cost and the new replica and host values from all the steps in the three WorkVar traces.

Some cases have the quartiles equal to the median and show no box (e.g. the data for P-Rwd in Figure 8(b) and P-PenRep in Figure 8(c)) and in some cases the quartiles and median are all zero. Where the median bar is not shown it is in all cases at the bottom of the box.

The box plots show that:

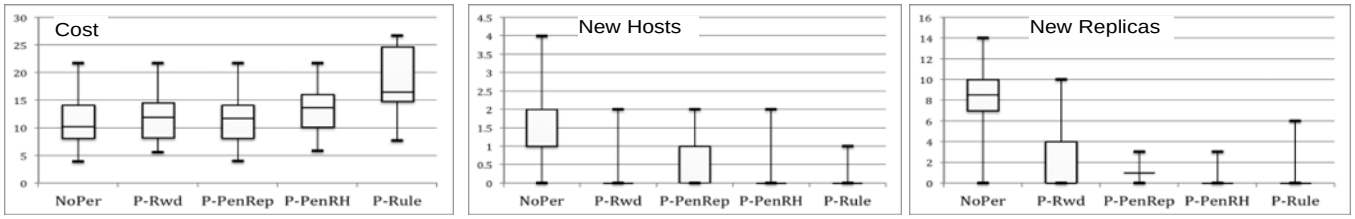
- *P-Rwd* has good median behavior (low running and reconfiguration costs) but a high peak-to-median ratio for new replicas (see Figure 5(c)). This indicates bursts of changes due to a buildup of sub-optimal decisions, which is undesirable.
- *P-Rule* gives much higher running costs.

Between *P-PenRep* and *P-PenRH*:

- *P-PenRep* gives lower running costs.
- *P-PenRH* gives much lower median reconfiguration costs, but with similar peak values.

This comparison appears to favour *P-PenRH*, however if the cost weighting were shifted to emphasize reduced running costs then *P-PenRep* might be preferred.

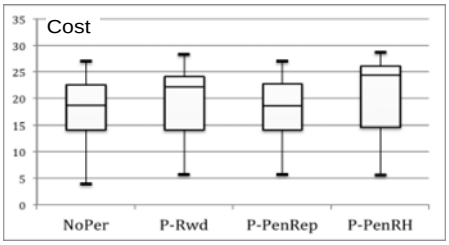
The tradeoff between running costs and startup costs is revealed in Figure 9, which plots one cost against the other for the different strategies, using values normalized against the values from *NoPer*. *P-Rule* gives the highest running cost and lowest startup cost, with *NoPer* is at the opposite extreme, as expected. *P-PenRH* and *P-PenRep* are low in both measures, with *P-PenRep* shifting the balance towards lower running costs and higher new hosts/replicas. *P-Rwd* is near the other two, and could be a contender. Figure 9 answers questions 1 and 2 from Section VI.A. It shows that persistence strategies *P-Rwd*, *P-PenRep*, and *P-PenRH* give cost close to optimal and give very similar results for both running cost and new replicas; P-PenRep is less effective at reducing new hosts (since it doesn't penalize them). P-Rule gives higher running costs.



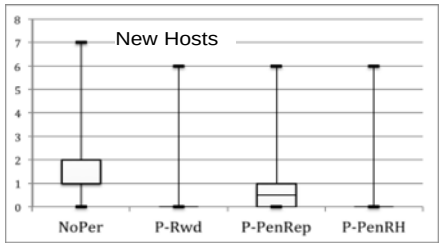
(a) WorkVar running cost

(b) WorkVar new hosts per step

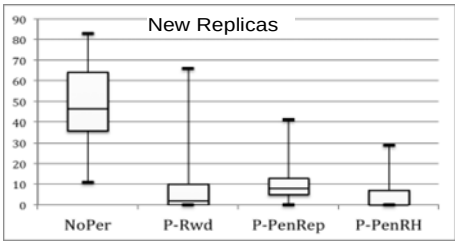
(c) WorkVar new replicas per step



(d) AppVar running cost

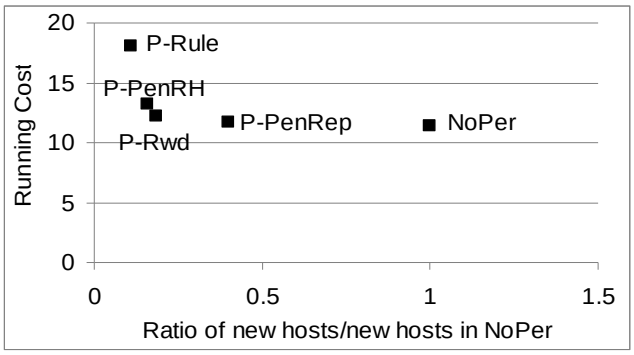


(e) AppVar new hosts per step

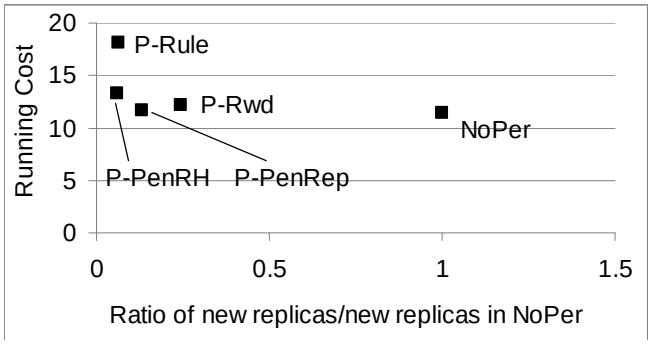


(f) AppVar new replicas per ste

Figure 5 Box-Whisker Plots for the WorkVar and AppVar Experiments



(a) against startup costs for new hosts



(b) against startup costs for new replicas

Figure 9 Normalized running costs vs startup costs, for the WorkVar experiments

C. AppVar: Application Addition and Removal

In the second group of experiments there are multiple applications, which are added and removed at random. This stresses the persistence strategies in a different way, since there are unavoidable new host and replica creations, and to achieve resource sharing between applications requires more frequent adjustments.

As before, the applications are created from the template in Figure 1, with randomly chosen parameters. Every application is different, so an application is not deleted and then added again later. The initial user populations were chosen to give a 0.1 stress rate on the initial deployment. The host pool had 22 hosts of each of the five types shown in Table 3.

P-Rule sometimes fails to find feasible solutions within the capacity of the host pool, given the randomly chosen parameters, so it is not included in the AppVar experiments.

The varying number of applications in the AppVar trace is shown by the cross-hatched bars in Figure 10. The increase or reduction in the number of applications in each interval is shown by the short black bars. Additions and removals occur in alternating intervals. Figure 11 shows a trace of the running costs found by the four strategies. The number of applications is shown by vertical bars using the axis labels on the right-hand side, while the points show the costs, using axis labels on the left-hand side. *NoPer* and *P-PenRep* have the lowest costs, and occasionally *P-PenRep* is the best. *P-PenRep* may be best due to the bin-packing heuristic selecting different hosts for the MIP optimization, giving different solutions.

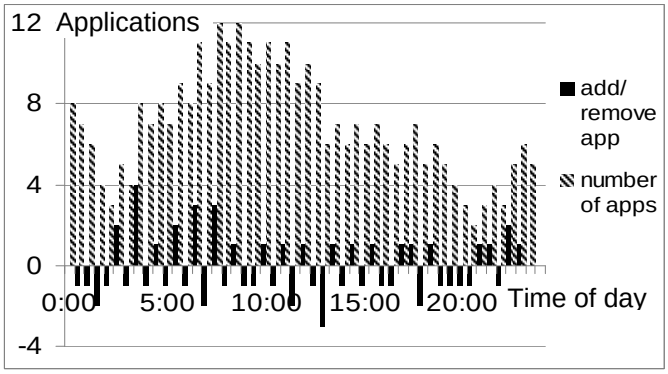


Figure 10 Variation of applications in AppVar

The second group (*P-Rwd* and *P-PenRH*) has higher running costs. *P-PenRH* is naturally the worst because it is focused on reducing startups, and gives lower priority to reducing running cost.

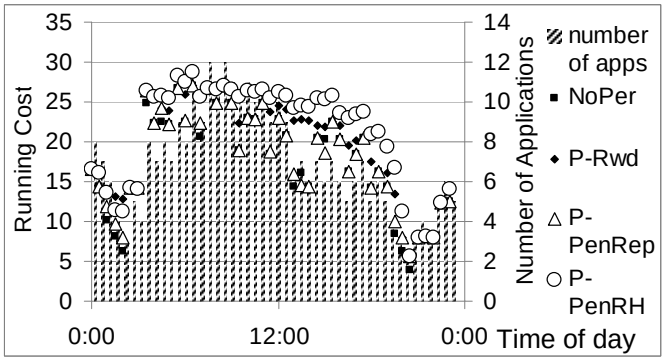


Figure 11 Running cost for AppVar

The reconfiguration cost results provide very similar comparisons to WorkVar, and are not shown for reasons of space. The box-whisker plots in Figure 8(d)-(f) confirm these comparisons:

- *P-PenRep* has the lowest running costs of the persistence strategies
- *P-PenRH* has lower reconfiguration costs and a better peak-to average ratio (less bursty behaviour)
- *P-Rwd* has high peak reconfiguration costs

Thus the AppVar results confirm our conclusions about questions 1 and 2 for a very different pattern of changes in the deployment.

D. Additional Analyses

A number of additional analyses were made on the data.

Degradation: Part of Question 2 asks if an accumulation of sub-optimal decisions degrades the running cost over time. The data was re-examined by dividing the running cost by the running cost of *NoPer*, and looking for values above 1.0.

In summary: for *P-PenRep* and *P-Rwd* the quotient was never more than 1.2 and showed no trend. For *P-PenRH* the quotient gradually rose to 1.5 and for *P-Rule* it rose to 2.5.

The results are shown for WorkVar_soccer, and for AppVar. From Figures 12 and 13 it seems clear that bad decisions do accumulate for *P-Rule*, which is not surprising since *P-Rule* never cleans up old decisions with a full

optimization. *P-Rule* is in any case not really a candidate for the best strategy. A slight tendency towards accumulation may be shown in Figure 13 for *P-Rwd* and *P-PenRH*, but the evidence is weak. *P-PenRep* seems to be immune from this problem. These results slightly favor *P-PenRep*.

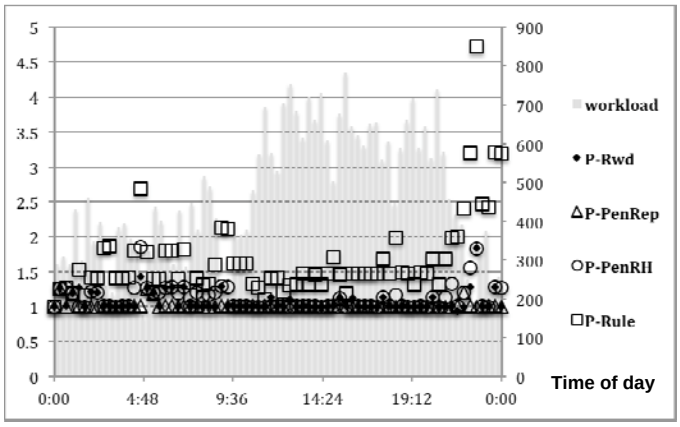


Figure 12 Running cost of persistence strategies divided by the value for *NoPer*, for WorkVar_soccer. The workload scale is at the right.

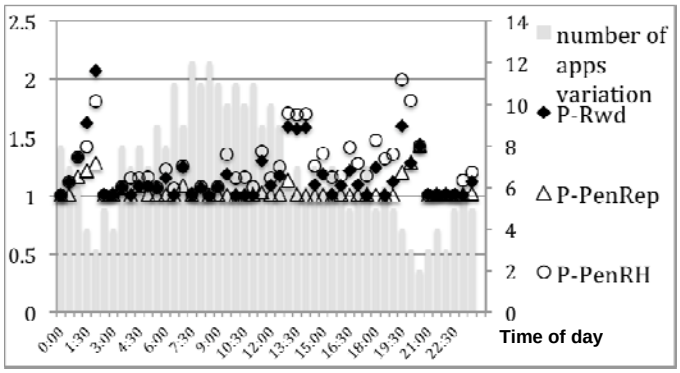


Figure 13 Running cost of persistence strategies divided by the cost for *NoPer*, for AppVar. The number of applications is at the right.

Large vs small changes: It may be that one strategy copes better with small changes and a different one is better for large changes. If so, one could select a strategy dynamically, depending on the observed change. The results for running costs and startups were examined for subgroups of steps with large and small changes of the workload or the applications, as shown in Tables 3 and 4. In summary, there was no effect on order of the strategies from best to worst, determined above.

The evaluation divided all the steps in the WorkVar traces into four classes, with changes in the number of users in the ranges (<-30), (-30, -1), (0, 30), (>30). Table 3 gives the costs and new entity counts by class. For the AppVar trace four classes were chosen with the change are in the number of Applications, in the ranges (≤-2), 1, 1, (≥2). Average values for total cost for these classes are shown in Table 4.

The Tables show that steps with increases in load have higher costs than those with decreases, which is not unexpected. There is however no visible difference in the relative performance of the strategies. Compared to *P-Rwd* and *P-PenRep*, *P-PenRH* never has higher reconfiguration cost or requires more new hosts or new replicas, and its "extra replicas" are always less, over all classes of step. However *P-PenRH* is sometimes worse for running cost.

Table 3 Effects of the five strategies, by class of step, for all the WorkVar traces considered together

Strategy	Class of Step: Change in Total Users			
	<-30	(-30, 0)	(0, 30)	>30
Running costs				
NoPer	10.34	12.45	10.54	12.78
P-Rwd	11.08	13.45	11.31	13.20
P-PenRep	10.49	12.69	10.76	13.09
P-PenRH	12.17	14.65	12.89	13.40
P-Rule	17.45	20.31	17.37	17.07
Average new hosts per step				
NoPer	1.30	1.40	1.48	2.26
P-Rwd	0	0	0.18	0.88
P-PenRep	0.23	0.48	0.47	1.29
P-PenRH	0	0	0.10	0.79
P-Rule	0	0	0.13	0.33
Average new replicas per step				
NoPer	8.23	8.15	7.10	9.33
P-Rwd	1.85	0.68	1.72	3.71
P-PenRep	0.83	0.76	0.70	1.69
P-PenRH	0	0	0.27	1.5
P-Rule	0	0	0.38	1.26

Table 4 Average total cost per step in AppVar cases

Strategy	Change in number of applications			
	≤ -2	-1	1	≥ 2
NoPer	153.54	155.74	185.43	226.50
P-Rwd	42.15	19.06	59.25	142.04
P-PenRep	31.22	32.63	62.37	116.61
P-PenRH	21.53	20.06	45.22	90.13

Penalty coefficients: If the values of *C_host* and *C_rep* used in *P-PenRH* are too small relative to the running costs then persistence might be ineffective. It is instructive to consider the impact of these coefficients on the overall Cost, through the ratio

$$\rho_2 = (\textit{Penalty/Running Cost}) \qquad (25)$$

which can be determined from a preliminary optimization trial. For the Penalty function to have an influence on the optimization, the value of ρ_2 at the optimum should not be much larger than unity. To adjust ρ_2, *C_host* and *C_rep* can be scaled together, maintaining the ratio ρ_1 = *C_host*/*C_rep* which is determined from the time delay to boot up a new host, versus the time to deploy a new task replica (the values 5 and 3 were chosen on this basis, in Section VI.A).

To examine this question, some of the experiments were re-run with the values of *C_host* and *C_rep* reduced by factors of 0.6 and 0.2. The results, including the values achieved for r2,

are shown in Table 5. In summary, for 0.6 there was no change in running cost or startups, but for 0.2 the running cost decreases by about 20% and startups increase by about 50% for AppVar and 20% for WorkVar. Persistence is thus somewhat less effective when the penalty coefficients are reduced five-fold.

These results show that the optimization is not very sensitive to ρ_2, but they also support choosing a target value of unity. A larger value of ρ_2 is satisfactory for the WorkVar cases (up to about 8) but not so good for the AppVar cases.

Table 5 The impact of startup penalty coefficients on the results for *NoPer* and *P-PenRH*, displayed as:

(*Running Cost*/mean new hosts/mean new replicas/$\boldsymbol{\rho_2}$)

	Percentage applied to (*C_host, C_rep)*		
Strategy	100% (=5, 3)	60% (=3, 1.8)	20% (=1, 0.6)
WorkVar results			
NoPer	11.43/ 1.62/ 8.11		
P-PenRH	13.3/ 0.26/ 0.49 / **4.8**	13.4/ 0.23/ 0.51 / **8.3**	11.4/ 0.38 /0.78 / **13.4**
AppVar results			
NoPer	17.53/ 1.81/ 49.57		
P-PenRH	21.0/ 0.45/ 5.94 / **1.05**	22.1/ 0.32/ 5.81 / **1.93**	18.3 /0.64/ 6.70 / **3.92**

Execution time: The execution times of the different strategies of Dynamic CloudOpt in the WorkVar and AppVar experiments were compared. The reconfiguration interval in the experiments is 20 minutes for WorkVar and 30 minutes for AppVar, suggesting that the optimization calculation should be completed in a time of the order of a minute or two.

The strategies take about the same time as *NoPer* (mostly within a factor of 2 either way) Tables A5 and A6 in Appendix VI-7 show there is no particular pattern. The average execution time per step increases for larger workload (in WorkVar) or number of applications (in AppVar). Apart from *P-Rule* the largest average time for a WorkVar case (for nearly 1000 users) is less than 1 s. For AppVar the largest average execution time (which occurred for 10-12 applications and 85 replicas) was just under a minute, suggesting that this is around the present scalability limit. *P-Rule* took up to 10 s. for the same cases.

The impact of persistence on scalability due to the number of applications can be examined in the AppVar experiments. For *NoPer* the steps with the most applications (10-12) took about 50 times longer than those with the least (2-3); for *P-PenRH* and *P-PenRep* the ratio was only about 30 times. Thus *P-PenRH* and *P-PenRep* appear to be at least as scalable as *NoPer*, and perhaps more scalable.

E. Overall Evaluation

The results above give these conclusions about the effectiveness of the four persistence strategies:

1. *P-Rwd* gives relatively low *Running Cost* when changes are small, but is more costly for large changes in the workload. *P-Rwd* is always substantially better than *NoPer*, however, so it is an effective approach to persistence.
2. *P-Rule* is effective only when loads are increasing, especially during spikes. It has relatively high *Running Cost* under decreasing workloads (shown in Appendix Table A2 for large negative changes). Presumably this is because some running tasks and hosts are not terminated.
3. *P-PenRep* is effective in controlling the number of changes within a specified range (set here at *frac* = 20% of existing replicas and hosts), in all experiments. Its behaviour is stable for either increasing or decreasing workloads.
4. *P-PenRH* has the lowest reconfiguration costs, but incurs somewhat higher *Running Cost*.

The answers to the five evaluation questions posed in Section VI.A can be summarized as follows:

1. Persistence is effective in reducing the reconfiguration effort by about 90%, with about a 5%-10% penalty in running costs. Of the strategies tested here,
   - *P-PenRep* appears to be the best, based on its low running costs which continue to be low over time,
   - *P-PenRH* is almost as good,
   - *P-Rwd* is erratic, with large peak reconfiguration efforts (shown in Figures 5(c) and 5(f)) despite a moderate average *Running Cost* overall,
   - *P-Rule* gives greatly increased running costs (about 50% above *NoPer*, see Figures 3 and 5(a)).

   Overall *P-PenRep* and *P-PenRH* are both winners, with *P-PenRH* favouring smaller algorithm execution times and reconfiguration changes of the solution, and *P-PenRep* giving lower running costs and being more stable. *P-Rwd* might be used occasionally, to cope with a large load increase.
2. The running cost with persistence is less than 10% higher than for *NoPer* and for P-Rwd, P-PenRep and P-PenRH it does not degrade markedly with time.
3. The change in the workload increases the reconfiguration effort for large positive changes, but the relative merit of the strategies is exactly the same for different positive and negative changes.
4. The value of *C_host* and *C_rep* relative to the running costs represented by the ratio ρ_2= *Penalty/Running Cost* makes little difference over the range $\rho_2 = (1, 9)$.
5. The persistence strategies take about the same time to execute as *NoPer*. The optimization time approaches one minute in the AppVar steps with 10-12 applications, and 80-85 replicas, so scalability does not extend to much larger problems.

P-PenRep has been configured here with a very large penalty on new replicas beyond the permitted fraction *frac*. With a smaller penalty it might approach closer to *NoPer* (more new replicas and lower running cost). However there does not seem to be any advantage in increasing the reconfiguration effort, since the running cost is already close to that of *NoPer*.

VII. CONCLUSIONS

A persistence strategy is successful for deployment

optimization in CloudOpt, using constraints or penalties. The experiments show that persistence reduces the reconfiguration effort (startups of hosts and replicas) to less than 20% of the non-persistent solutions, with a small increase (less than 10%) in the running cost. The strategies can deal with recovery from host failures in a straightforward way, doing the optimization with a modified host pool.

The evaluation suggests that the two strategies *P-PenRep* (which penalizes new replicas at any step, that exceed a fraction of the existing replicas) and *P-PenRH* (which penalizes all new hosts and new replicas) are both good, with *P-PenRH* favouring smaller algorithm execution times and reconfiguration changes of the solution, and *P-PenRep* giving lower running costs and being more stable.

The strategies appear to be rather insensitive to the strategy parameters for penalty costs, which is a good thing. A method was described for choosing the penalty weights, taking advantage of this insensitivity.

Scalability is important for clouds. The examples above show that CloudOpt is practical when applied to a few applications, totaling up to about 10 (see Figure 10), each with seven deployable units corresponding to the seven tasks in Figure 3. For large number of applications, CloudOpt can be applied to groups of them, of a size totaling up to about 50 or 70 deployable units. A previous study on an earlier version of CloudOpt [17] showed that the advantage of sharing among applications (compared to separate optimization of each application) leveled off at a cost improvement of about 20% above 10 applications, based on studies using the same deployable units as shown in Figure 3. Thus it is not necessary to consider everything in the cloud together in one giant optimization problem, to reap most of the benefit. Considering small groups of applications allows the parallel optimization of all groups, which can be scaled indefinitely.

The scalability of a single optimization calculation is not made worse by including persistence (see the Running Time discussion in Section VI.D). It is also interesting that the scalability of CloudOpt is limited by the MIP solver, and not by the nonlinear performance model. With a different MIP solver, larger scales may be accessible.

The problem considered here can be broadened. The dynamic algorithm as it is already accommodates host and software failures and repairs, even though they were not included in the experiments reported. Communications delays which are the same for every host pair can be included in the performance model as a constant total delay for each response. However heterogeneous communications delays, between racks and between clusters, will require a more complex approach and are the subject of current work.

REFERENCES

[1] Arlitt, M., Jin, T., "Workload Characterization Of The 1998 World Cup Web Site", Hewlett-Packard Labs, Technical Report HPL-99-35R1, Sept. 1999.

[2] Bertini, L., Leite, J.C.B., Moss, D., "Power Optimization For Dynamic Configuration In Heterogeneous Web Server Clusters", *J. Systems and Software.* V. 83, no. 4, pp. 585-598, April 2010.

[3] Bilgin, S. and Azizoğlu, M.,"Operation Assignment And Capacity Allocation Problem In Automated Manufacturing Systems", *J. of Computers and Industrial Engineering*, V. 56, no 2, pp. 662-676, Mar 2009.

[4] Bokhari, S. H. "Partitioning Problems in Parallel, Pipeline, and Distributed Computing", *IEEE Trans Computers*. V.37, no. 1, pp. 48-57, Jan 1988.

[5] Bouchenak, S.; De Palma, N., Hagimont, D., Krakowiak, S., Taton, C.; "Autonomic Management of Internet Services: Experience with Self-Optimization", *Proc Int Conf on Autonomic Computing,* (ICAC '06), June 2006, pp 309 - 310.

[6] Brown, G., Dell, R., Wood, K., "Optimization and Persistence", *Interfaces,* V. 27, pp. 15-37, 1997.

[7] Carrera, D., "Adaptive Execution Environments for Application Servers", *PhD dissertation*, *Tech Univ of Catalonia,* 2008.

[8] Chaisiri, S.; Bu-Sung Lee; Niyato, D., "Optimal Virtual Machine Placement Across Multiple Cloud Providers", in *Proc IEEE Asia-Pacific Services Computing Conf,* pp. 103 – 110, Dec 2009.

[9] Chao, S., Chinneck, J.W., Goubran, R.A., "Assigning Service Requests in Voice-over-Internet Gateway Multiprocessors", *Computers and Operations Research*, V. 31, pp. 2419-2437, 2004.

[10] Coffman, E.G, Garey, M.R., Johnson, D.S., "An Application Of Bin-Packing To Multiprocessor Scheduling", *SIAM Journal on Computing*, V. 7, pp. 1-17, Feb. 1978.

[11] Franks, G., Al-Omari, T., Woodside, M., Das, O., Derisavi, S., "Enhanced Modeling and Solution of Layered Queueing Networks", *IEEE Trans on Software Engineering,* V.35, no 2, pp.148-161, Mar. 2009.

[12] Hellerstein, J. L., Diao, Y., Parekh, S., Tilbury, D. M. "Feedback Control of Computing Systems", Wiley, 2004.

[13] Isard, M, Prabhakaran, V, Currey, J, Wieder, U, Talwar, K. Goldberg,A, "Quincy: Fair Scheduling for Distributed Computing Clusters", *Proc. ACM Symp. on Operating Systsem Principles*, 2009.

[14] Kansal, A., Zhao, F., Liu, J., Kothari, N., Bhattacharya, A., "Virtual Machine Power Metering And Provisioning", *Proc 1st ACM Symposium on Cloud computing (SoCC '10)*, 2010.

[15] Karve, A., Kimbrel, T., Pacifici, G., Spreitzer, M., Steinder, M., Sviridenko, M., Tantawi, A., "Dynamic Placement For Clustered Web Applications", in *Proc 15th Int Conf on World Wide Web*, Edinburgh, May 23 - 26, 2006.

[16] Kasbekar, M., "On Efficient Delivery of Web Content", keynote to SIGMETRICS 2010, published at http://www.sigmetrics.org /sigmetrics2010/greenmetrics/MangeshKasbekar.pdf. Accessed Feb. 2012

[17] Li, J., Chinneck, J., Woodside, M., Litoiu, M., Iszlai, G, "Performance Model Driven QoS Guarantees and Optimization in Clouds", in *Proc Workshop on Software Engineering Challenges in Cloud Computing @ ICSE 2009*, Vancouver, May 2009.

[18] Li, J., Chinneck, J., Woodside, M., Litoiu, M, "Fast Scalable Optimization to Configure Service Systems having Cost and Quality of Service Constraints", *Proc 6th Int Conf on Autonomic Computing* (ICAC.09), Barcelona, June 2009.

[19] Li, J., Chinneck, J., Woodside, M., Litoiu, M, "Deployment of Services in a Cloud Subject to Memory and License Constraints", in *Proc 2nd IEEE Intl Conf on Cloud Computing* , Bangalore, India, September 21-25, 2009.

[20] Li, J., "Fast Optimization for Scalable Application Deployments in Large Service Centers", *PhD thesis*, *Carleton University*, May 2011.

[21] Li, J., Chinneck, J., Woodside, M., Litoiu, M., "CloudOpt: Multi-Goal Optimization of Application Deployments across a

Cloud", *7th Int. Conf. on Network and Service Management (CNSM 2011)*, October 24-28 2011, Paris, France

[22] Li, L., Tian, R., Yang, B., Gao, Z., "A Model of Web Server's Performance-Power Relationship", in *Proc Int Conf on Communication Software and Networks 2009*

[23] Litoiu, M., Rolia, J., Serazzi, G, "Designing Process Replication and Activation: A Quantitative Approach", *IEEE Trans on Software Engineering,* V. 26, No. 12, pp. 1168-1178, Dec. 2000.

[24] Menascé, D. A., Bennani, M.N., "Dynamic Server Allocation for Autonomic Service Centers in the Presence of Failures". in *"Autonomic Computing: Concepts, Infrastructure, and Applications",* eds. S. Hariri and M. Parashar, CRC Press, 2006.

[25] Menascé, D. A., Casalicchio, E., Dubey, V., "A Heuristic Approach To Optimal Service Selection In Service Oriented Architectures", in *Proc 7th International Workshop on Software and Performance,* Princeton, USA, Jun 2008.

[26] Petrucci, V., Loques, O., Moss, D., "A Dynamic Optimization Model For Power And Performance Management Of Virtualized Clusters", in *Proc 1st Int Conf on Energy-Efficient Computing and Networking (e-Energy '10)*, Passau, Germany, April 2010.

[27] Polo, J., Castillo, C., Carrera, D., Becerra, Y., Steinder, M., Whalley, I., Torres, J., Ayguadé, J., "Resource-aware Adaptive Scheduling for MapReduce Clusters". in *Proc 12th Int Middleware Conference*. Lisbon, Portugal. Dec 2011.

[28] Rolia, J. A., Sevcik, K. C., "The Method of Layers", *IEEE Trans on Software Engineering*, Vol. 21, No.8,pp. 689-700, Aug 1995.

[29] Soror, A. A., Minhas, U. F., Aboulnaga, A., Salem, K., Kokosielis, P., Kamath, S, "Automatic Virtual Machine Configuration For Database Workloads", in *Proc SIGMOD 2008*, Vancouver, Canada, June 09 - 12, 2008.

[30] Steinder, M., Whalley, I., Carrera, D., Chess, D, "Server Virtualization In Autonomic Management Of Heterogeneous Workloads", *Proc. Integrated Management 07 ,* May 2007.

[31] Tang, C., Steinder, M., Spreitzer, M., Pacifici, G, "A Scalable Application Placement Controller For Enterprise Data Centers", in *Proc 16th Int Conf on the World Wide Web*, Banff, 2007.

[32] Toktay, L.B., Uzsoy, R., "A Capacity Allocation Problem With Integer Side Constraints", *European J. of Operational Research*, V.109, Issue.1, pp.170-182, 1998.

[33] Vengerov, D., "A Reinforcement Learning Approach To Dynamic Resource Allocation", *J. of Engineering Applications of Artificial Intelligence*, V.20, No.3, pp.383-390, April 2007.

[34] Verma, A., Cherkasova, L., Campbell R.: "ARIA: Automatic Resource Inference and Allocation for MapReduce Environments" *Proc 8th IEEE Int Conf on Autonomic Computing (ICAC'2011)*, June 14-18, 2011, Karlsruhe

[35] Woodside, C.M., Monforton, G.G., "Fast Allocation of Processes in Distributed and Parallel Systems", *IEEE Trans on Parallel and Distributed Systems*, V. 4, No. 2, pp. 164-174, 1993.

[36] Woodside, C.M., Neilson, J. E., Petriu, D. C., and Majumdar, S., "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software", in *IEEE Trans on Computers.* 44, 1, pp.20-34, Jan. 1995.

[37] Xu, W., Zhu, X., Singhal, S., Wang, Z., "Predictive Control for Dynamic Resource Allocation in Enterprise Data Centers", Proc. 10th Network Operations and Management Symposium, 2006.

[38] Zhang, M., Martin, P., Powley, W., Bird, P., "Using Economic Models To Allocate Resources In Database Management Systems", in *ProcCASCON 2008*, Toronto, October 2008.

[39] Zheng, T., "Model-based Dynamic Resource Management for Multi Tier Information Systems", *PhD thesis*, *Carleton Universtiy*, August 2007.

[40] Zheng, T.,Woodside, C. M., Litoiu, M., "Performance Model Estimation And Tracking Using Optimal Filters", *IEEE Trans on Software Engineering,* Vol. 34 , No. 3, pp. 391-406, 2008.