Mistral: Dynamically Managing Power, Performance, and Adaptation Cost in Cloud Infrastructures

Gueyoung Jung† Matti A. Hiltunen‡ Kaustubh R. Joshi‡
Richard D. Schlichting‡ Calton Pu†

†College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
{gueyoung.jung,calton}@cc.gatech.edu

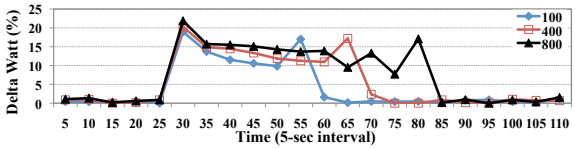
‡AT&T Labs Research
180 Park Ave.
Florham Park, NJ, USA
{kaustubh,hiltunen,rick}@research.att.com

***Abstract*—Server consolidation based on virtualization is a key ingredient for improving power efficiency and resource utilization in cloud computing infrastructures. However, to provide satisfactory performance in such scenarios under changing application workloads, dynamic management of the consolidated resource pool is critical. Unfortunately, this type of management is also challenging in cloud platforms because of the inherent tradeoffs between power and performance, and between the cost of an adaptation and its benefit. In this paper, we present Mistral, a holistic controller framework that optimizes power consumption, performance benefit, and the transient costs incurred by various adaptations and the controller itself to maximize overall utility. Mistral can handle multiple distributed applications and large-scale infrastructures through a multi-level adaptation hierarchy and scalable optimization algorithm. Through extensive experiments, we show that our approach outstrips other strategies, each of which represents addressing the tradeoff between only two objectives among power consumption, performance, and transient costs.**

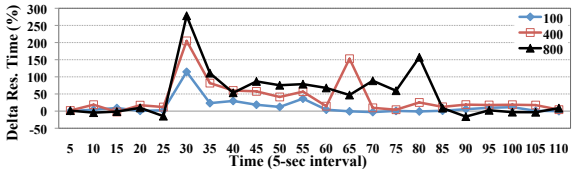
I. Introduction

Improved resource utilization by dynamically taking advantage of workload variations is a key differentiator of shared infrastructure approaches such as cloud computing. Through actions such as virtual machine (VM) migration and resource capping, cloud infrastructure providers not only accommodate demand spikes by temporarily taking resources away from underutilized applications, but also save energy by shutting down idle resources. However, consolidation can also have a detrimental impact on application performance, and thus, must be used very carefully in a wide array of distributed online services such as online shopping, communications, and enterprise applications where savings cannot come at the cost of a degraded user experience.

In addition to the power-performance tradeoff that is inherent in server consolidation, infrastructure providers must also consider adaptation benefit and cost tradeoffs since workload varies dynamically, and runtime consolidation actions such as migration are not free. While VM technology has made great strides in reducing the downtime during migration to a few hundred milliseconds (e.g., [1]), the end-to-end performance and power consumption impacts can still be significant. For example, Figure 1 shows the increase in power consumption and end-to-end response time of a 3-tier Web/Java/MySQL application as a function of time during the live migration of a single of its Xen-based VMs (initiated at the 25sec mark). The measurements, shown for three different workloads of 100, 400, and 800 concurrent user sessions, indicate that the impact is not only significant, but that it depends on the workload and is incurred over a substantial period of time.



(a) Power consumption



(b) Response time

Fig. 1: Costs of a single VM live-migration

Coupled with a changing workload, adaptation costs can lead to complete rethinking of the best strategy for power savings. For example, when the workload is rapidly changing, it may be better to suffer a slight performance degradation rather than trigger an expensive migration whose costs may never be recouped before another adaptation is needed. Or, a cheap but modest change such as the redistribution of resources amongst VMs may be a more effective than powering up a new host. Finally, the power cost and decision delay incurred by the system making the adaptation decision must also be considered. Often it may be better to make a suboptimal decision quickly rather than invest time and energy searching for savings that are not enough to recoup the investment. Although there have been extensive prior studies both on

balancing steady-state performance and on minimizing the cost of VM migrations (see Section VI), we are not aware of any work that balances steady state performance and power with the dynamic adaptation costs under changing workloads.

In this paper, we present Mistral, a holistic optimization system that balances power consumption, application performance, and transient power/performance costs due to adaptation actions and decision making in a single unified framework. By doing so, it can dynamically choose from a variety of actions with differing effects in a multiple application, dynamic workload environment. We extend control techniques developed in our prior work [2] and make new contributions by incorporating the cost of power in both steady state and during adaptations, and enable significant power savings by migrating applications away from idle resources and shutting them down only when appropriate. Also for the first time, we provide the controller with an ability to factor its own power consumption and decision delays into its decision making. Finally, we construct the controller as a scalable, multi-level hierarchical system that can deal with a large number of applications and hosts, and also with adaptation actions at multiple time-scales ranging from a few milliseconds to tens of minutes. Finally, we show the benefits of our approach by extensive experimental comparisons to three alternative strategies, each of which represents optimizing the tradeoff between any two objectives among performance, power consumption, and transient costs.

II. Overview

Mistral controllers optimize over dual objectives of power and performance, and therefore, the framework uses a *utility* based model to compare both on a uniform footing. We begin by first describing this utility model and subsequently provide an overview of Mistral's architecture.

A. Assumptions

We assume a set of distributed applications $s \in S$ to be managed, each of which consists of multiple tiers of components (e.g., web, application, and database servers). Each tier may consist of several replicas, and these are hosted inside VMs running on the physical hosts, with one replica per VM for the sake of convenience. Each application is also associated with a set of transaction types (e.g., home, login, search, browse) through which users access its services. Each transaction type generates a unique call graph through some subset of application tiers. For example, a browse request invokes a web server forwarding the request to the application server, which makes several calls to the database. The workload w^s for each application s is then defined as a vector of the mean request rate for each transaction type, and overall workload for the entire system W as the vector of workloads for all hosted applications.

Mistral controllers are activated from time-to-time to determine which physical machine each VM should reside on, and how much CPU it should receive[1]. A *system configuration* c_i is represented by the set of VMs in the system, the physical machine on which they are hosted, and the CPU fraction allocated to them. CPU allocations are enforced using Xen's credit-based scheduler and can be changed, while VMs can be moved from one machine to another using live migration. As capacity demands change, component replicas can be added or removed from each application tier by migrating them from or to a cold-store repository respectively, while physical hosts can be turned on or off to save power. These actions form the set of adaptation actions $\mathcal{A}$. Each action a transforms the system from one configuration c_i to another c_{i+1}.

B. Utility

To decide $when$ and how to adapt at runtime, Mistral estimates the potential future benefit of each adaptation action a as well as its cost in terms of changes in power and performance utility values. The cost of each adaptation action a depends on its duration $d(a)$ and impact on response times and power consumption. Meanwhile, the benefit of adaptation depends on how long the system remains in the new configuration, defined as stability interval CW_i. Thus, the overall *system utility* U consists of the *power utility* U_{pwr} of the system after adaptation, the *application utilities* U^s_{RT} based on the performance of each application after adaptation, and transient *adaptation costs* $U_{RT}(c, a)$ incurred during adaptation.

To compute application utility, each application has its own performance objective in the form of a target mean response time $TRT^s(w^s)$ computed over an application defined monitoring window M, a reward $R(w^s)$ for meeting the target response time in a single monitoring period, and a penalty $P(w^s)$ for missing it. The response time targets, rewards, and penalties are allowed to depend on the request rate, thus allowing arbitrary application utility functions to be defined. As described in Section V, our experiments use a function in which the reward increases and the penalty decreases as the workload increases. Given the measured or predicted request rate for application s at time-step i as W^s_i, the system configuration as c_i, the measured or predicted mean response time RT^s_i, the target response time $TRT^s(W^s_i)$, reward $R^s(W^s_i)$, and penalty $P^s(W^s_i)$, the application utility accrual rate is given as:

$$U^s_{RT_i}(c_i) = \begin{cases} R^s(W^s_i)/M & \text{if } RT^s_i \leq TRT^s \\ P^s(W^s_i)/M & \text{otherwise} \end{cases} \quad (1)$$

For the (negative) utility accrued due to power consumption, we focus on power consumed by the physical hosts in this paper. While power consumed by cooling infrastructure is also a major concern in typical data centers, we do not consider it explicitly since cooling overheads can be approximately modeled as a fixed percentage of the power consumed by the computing infrastructure [3]. We convert the energy cost per

[1]We do not currently manage other resource types such as memory, network, or disk except to ensure that VMs are only hosted on machines that have sufficient memory to satisfy the VM's fixed memory requirements.

Watt-hour $PCWh$ to the instantaneous rate at which utility is accrued using the equation

$$U^s_{pwr}(c_i) = -pwr(c_i) \cdot PCWh \quad (2)$$

where $pwr(c_i)$ is the predicted or measured mean power consumption (in Watts) of the system in configuration c_i over the monitoring interval M.

To convert adaptation costs to a utility value, Mistral computes the instantaneous rate at which an application accrues utility during the execution of a series of adaptation actions in an interval. Let $RT^s(c_i, a)$ and $pwr(c_i, a)$ be the measured or predicted mean response time of application s and the power consumption of the system when adaptation action a is executed in configuration c_i. By plugging these values into Equations 1 and 2, the corresponding utility accrual rates $U^s_{RT_i}(c_i, a)$ and $U^s_{pwr}(c_i, a)$ during execution of action a starting from a configuration c_i can be computed.

Finally, we put together these components to obtain the overall utility accrued between two invocations of a Mistral controller. Let the initial configuration be c_i, and let CW_{i-1} be the stability interval as defined earlier. Let W^s_i represent the fixed workload during the stability interval. The stability interval ends when the workload deviates from a band of width B^s, called the *workload band*, centered around this fixed value i.e., $(W^s_i - B^s/2, W^s_i + B^s/2)$. When that happens, the controller is invoked to evaluate the need for adaptation and may execute a sequence of adaptation actions $A_i = a_1, a_2, \ldots a_n$ to transform c_i into a new configuration c_{i+1}. We anticipate that this new configuration is retained until the end of the new stability interval CW_i. Let $d(a_1), d(a_2), \ldots, d(a_n)$ be the length of each adaptation action, and let $c^1, c^2, \ldots, c^n$ be intermediate configurations generated by applying the actions starting from c_i. Let c^0 be the initial configuration c_i and c^n be the final configuration c_{i+1}. Then, the overall utility is given by

$$U_i = \sum_{a_k \in A_i} d(a_k) \cdot \left(U^s_{pwr}(c^{k-1}, a_k) + \sum_{s \in S} U^s_{RT_i}(c^{k-1}, a_k)\right) + \left(CW_i - \sum_{a_k \in A_i} d(a_k)\right)\left(U^s_{pwr}(c^{i+1}) + \sum_{s \in S} U^s_{RT_i}(c_{i+1})\right) \quad (3)$$

The first term sums the system-wide power utility and application specific performance utilities accrued during each adaptation action execution (i.e., the action costs), while the second term sums the power and application utilities of the resulting configuration c_{i+1} until the end of the stability interval. By maximizing this utility, Mistral can balance the cost accrued over the duration of an adaptation with the benefit accrued between its completion and the next adaptation.

C. Architecture

In a large data center environment, Mistral is deployed in the form of a multi-level hierarchical control scheme with multiple instances of Mistral controllers managing different subsets of hosts and applications and operating at different timescales. The lowest level controllers manage a small number of machines and the applications that are hosted on them (e.g., a single rack). At the next higher level, a controller manages machines owned by multiple lower level controllers.

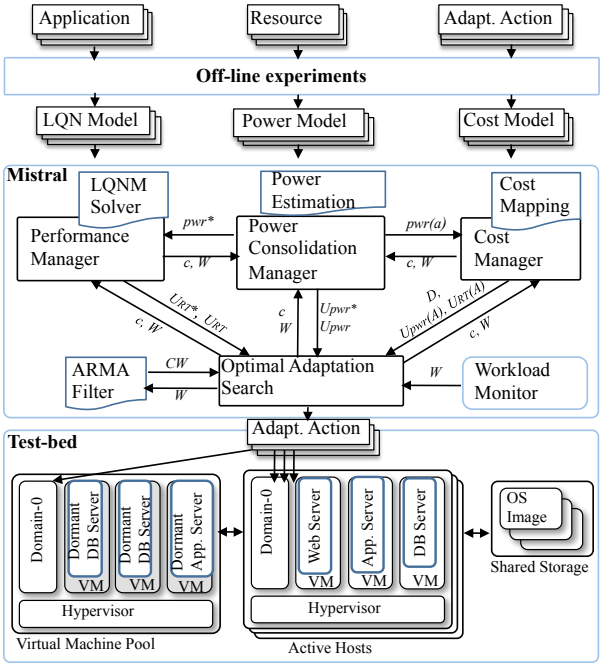


Fig. 2: Architecture of a Single Mistral Controller

To understand how the controllers interact, consider that an adaptation action is only chosen by a controller if it is anticipated to increase utility over the next CW_i time units. From equation 3, it can be seen that as the stability interval becomes longer, adaptation is less frequent, but the benefits of adaptation can accrue for longer periods. Thus, longer stability intervals make increasingly disruptive actions with potentially more significant benefits (e.g., VM live migrations, power cycling) worthwhile, while short stability intervals may rule out all but the quickest actions (e.g., CPU capacity tuning). Stability intervals can be made longer by making the workload bands wider, i.e., allowing a larger change in workload before adaptation is needed. Therefore, lower level controllers are configured with very narrow workload bands. They may be invoked very rapidly, but only produce modest changes, which coupled with their limited target domain, ensure quick decision times. Higher level controllers have increasingly larger workload bands, longer times between invocation (e.g., hourly, daily, weekly), larger sets of more potent actions to choose from, more hosts and applications to consider, and correspondingly take longer to make their decisions. Our experimental evaluation uses a two-level hierarchy with the lower level controller invoked periodically once every unit interval (i.e., the band is 0), while the high level controller uses a band of 8 req/sec.

Figure 2 illustrates the architecture of a single controller. The architecture consists of a set of "predictor modules" and an "optimizer module". The predictor modules, which include the Performance Manager, the Power Consolidation Manager, the Cost Manager, and the Workload predictor (ARMA filter), are described in Section III and use analytical models

to predict utility values of new configurations and actions being considered by the optimizer. Given a configuration c and workload W, the Performance and Power Consolidation Managers predict the corresponding application utility and power utility values. When provided with a list of actions in addition to c and W, the Cost Manager predicts the action costs, while the ARMA filter uses previous workload history to predict the stability intervals. The optimizer module consists of the Optimal Adaptation Search engine and is responsible for choosing the optimal set of actions that will maximize the utility. The search is guided by upper bounds of utility estimates (denoted by the superscript "*") which are provided by the predictor modules. The optimizer is described in Section IV.

III. Model Based Prediction

Next, we describe the queuing models used by the Performance Manager to predict application performance, the analytical models used by the Power Consolidation Manager to predict the overall system's power consumption, the measurement-based techniques used by the Cost Manager to predict transient adaptation costs, and the predictive filter used by the Workload predictor to predict how long the system workload will remain approximately at its current value.

A. Application Modeling

To estimate the benefit of a configuration, we need to estimate the response time of each application for a given workload. In this paper, we use layered queuing network (LQN) models from our prior work on performance optimization [4]. The tier servers are modeled as FCFS queues, while hardware resources (e.g., CPU, disk, and bandwidth) are modeled as processor sharing (PS) queues. Interactions between tiers triggered by an incoming transaction are modeled as synchronous calls in the queuing network and our models also account for the resource sharing overhead imposed by Xen [5]. The models are also used to estimate the CPU utilization of each tier for a given workload. The parameters for models (e.g., per-transaction service time at each hardware resource) are measured in an offline measurement phase. In this phase, each application is instrumented at each tier using system call interception. Then, delays between incoming and outgoing messages are measured per transaction. More details can be found in [4].

B. Power Consumption Modeling

To estimate power consumption of a configuration and workload, we employ a utilization-based model used in previous studies (e.g., [3], [6]). Specifically, for a physical machine, we use an empirical non-linear model, $pwr = pwr_{idle} + (pwr_{busy} - pwr_{idle}) * (2\rho - \rho^r)$, where pwr_{idle} represents the power consumption of the machine at standby state and pwr_{busy} represents the maximum power consumption of the machine observed in our scenario, and ρ is the CPU utilization of the machine estimated by the LQN models at the current workload. A tuning parameter r is used to minimize the square error and is obtained at a model calibration phase. We use offline experiments to calibrate the non-linear model to fit into actual power consumption observed using a power meter. The total power usage of the system is simply the sum of physical machines' power usages.

C. Transient Adaptation Costs

In this paper, we consider six adaptation actions: increase/decrease a VM's CPU capacity by a fixed amount, addition/removal of a VM, live-migration of a VM between hosts, and shutting down/restarting physical hosts. Addition of a VM replica is implemented by migrating a dormant VM from a pool of VMs to the target host and activating it by allocating CPU capacity. A replica is removed by migrating it back to the pool.

Costs of these adaptation actions are measured experimentally offline for different workloads and configurations and are stored in tables used at runtime. Specifically, we measure the following attributes of each adaptation: (a) adaptation duration, (b) change in response time for the application being adapted as well as applications co-located with the application being adapted, and (c) change in power consumption during the adaptation.

We use the following process to measure these costs. For each adaptation action a, we set up a target application s along with a background application s' such that all replicas from both applications are allocated equal CPU capacity (40% in our experiments). Then, we run multiple experiments, each with a random placement of all VMs across all the physical hosts. During each experiment, we subject both the target and background application to the workload W^s and $W^{s'}$, and after a warm-up period of 1 minute, measure response times of two applications RT^s and $RT^{s'}$ and the total power usage of corresponding physical machines pwr. Then, we execute the adaptation action a, and measure the duration of the action, $d(a)$, the response time of each application during adaptation, $RT^s(a)$ and $RT^{s'}(a)$, and the power usage on affected physical machines $pwr(a)$. If none of application's VMs are co-located with the VM impacted by a, no background application measurements are made. We use these measurements to calculate delta response times $\Delta RT^s(a) = RT^s(a) - RT^s$ and $\Delta RT^{s'}(a) = RT^{s'}(a) - RT^{s'}$, and the delta power usage $\Delta pwr(a) = pwr(a) - pwr$. These deltas along with the action duration are averaged across all random configurations, and their values are encoded in a cost table indexed by the workload. When Mistral requires an estimate of adaptation costs at runtime, it measures the current workload W and looks up the cost table entry with the closest workload W'.

D. Workload Prediction

Given that our approach balances immediate adaptation costs versus their potential future benefits, the ability to estimate workload changes is crucial. The approach we have chosen is to estimate how long the workload stays approximately at its current level based on the history of workload changes. To recall from Section II-B, the stability interval for

an application s at time t is the period of time for which its workload remains within $\pm b/2$ of the measured workload W_t^s at time t, where b is a user-specified threshold. This band [W_t^s - $b/2$, W_t^s + $b/2$] is called the *workload band* B_t^s. When an application's workload exceeds the workload band, Mistral must re-evaluate the system configuration. When the workload falls below the band, the controller must check if other applications might benefit from the resources that could be freed up.

At each unit monitoring interval i, Mistral checks if the current workload W_i^s is within the current workload band B_j^s. If one or more application workloads are not within their band, Mistral estimates a new stability interval CW_{j+1}^e for the next control window and updates the bands based on the current application workloads. To estimate the stability intervals, we employ an auto-regressive moving average (ARMA) filter commonly used for time-series analysis (e.g. [7]). The filter uses a combination of the last measured stability interval CW_j^m and an average of the k previously measured stability intervals to predict the next stability interval using the equation: $CW_{j+1}^e = (1-\beta) \cdot CW_j^m + \beta \cdot 1/k \sum_{i=1}^{k} CW_{j-i}^m$. Here, the factor β determines how much the estimator weighs the current measurement against past historical measurements. It is calculated using an adaptive filter to quickly respond to large changes in the stability interval while remaining robust against small variations. To calculate β, Mistral first calculates the error ε_j between the current stability interval measurement CW_j^m and the estimation CW_j^e using both current measurements and the previous k error values as $\varepsilon_j = (1-\gamma) \cdot |CW_j^e - CW_j^m| + \gamma \cdot 1/k \sum_{i=1}^{k} \varepsilon_{j-i}$. Then, $\beta = 1 - \varepsilon_j / \max_{i=0...k} \varepsilon_{j-i}$. This technique dynamically gives more weight to the current stability interval measurement by generating a low value for β when the estimated stability interval at time i is close to the measured value. Otherwise, it increases β to emphasize past history. We use a history window k of 3, and set the parameter γ to 0.5 to give equal weight to the current and historical error estimates.

IV. Optimization Approach

The holistic optimization approach of Mistral relies on a simpler optimizer, Perf-Pwr optimizer, to provide the best configuration while ignoring any adaptation costs. In this section, we will first describe Perf-Pwr optimizer and then the holistic algorithm of Mistral.

A. Perf-Pwr Optimizer

Perf-Pwr optimizer generates the optimal tradeoff between performance and power consumption for a given workload when any transient adaptation costs are ignored. Specifically, Perf-Pwr finds the optimal capacities of VMs that can be packed on as few server machines as possible while balancing performance and power usage. Similar to our prior study [4], Perf-Pwr optimizer employs a heuristic bin-packing algorithm to place given VMs to hosts and a classic gradient-based search algorithm, but extends the algorithm to deal with variable number of active host machines and their power consumption.

Perf-Pwr optimizer determines the optimal configuration (i.e., one that maximizes application utility) first for the whole system (all hosts active) and then reduces the number of hosts to see if a smaller number of active hosts would optimize overall utility (application utility + power usage). Specifically, for any given set of hosts, Perf-Pwr optimizer initially allocates maximum CPU capacities for all VMs. Then, it attempts to place (pack) these VMs on the given set hosts (bins). The bin-packing algorithm used by the optimizer chooses the host that has the largest space among used hosts. If no such host is found, it chooses a new empty host only if it is available. If the bin packing fails, the optimizer starts a search process where, in each iteration, it generates a set of candidate configurations by (a) reducing the capacity of individual VMs and (b) reducing the replication level of an application component (and thus, removing one VM). Then, it chooses the candidate that has the highest *gradient* value among all the candidates, where the gradient is defined as $\nabla\rho = (\rho^{new} - \rho)/(U_{RT}^{new} - U_{RT})$ and ρ^{new} and U_{RT}^{new} represent each new candidate's CPU utilization and performance utility, respectively. It attempts to pack the chosen candidate c_i on the given set of hosts and if the packing fails, the algorithm performs the next iteration using configuration c_i as the new starting point. If the packing succeeds, the optimizer considers the resulting configuration as a *potential optimal configuration* and repeats the search with a reduced number of hosts. For each potential optimal configuration, the optimizer estimates watts consumed by each host by summing all hosted VMs' utilization and also shared utilization (i.e., consumed by Dom-0). The optimizer stops reducing number of hosts when the number of hosts reaches a threshold that can host minimum capacities of the VMs. The potential optimal configuration that has the largest utility is chosen as the "ideal configuration" c^* and its utility denotes the "ideal utility" $U^* = U_{RT}^* - U_{pwr}^*$. The ideal utility is an upper bound for the holistic optimization since it ignores adaptation costs.

B. Holistic Optimization

The core of Mistral is a holistic optimization algorithm that incorporates power and performance overheads caused by adaptation actions and the cost of the decision making process into the tradeoff formulation. Given the utility function and models, Mistral determines the optimal sequence of adaptation actions that transforms the current configuration c to the new optimal configuration.

The algorithm constructs a search graph, where edges are adaptation actions and vertices are system configurations. The search starts from the vertex v_0 representing the current configuration c. A new configuration (vertex) at each depth of the search graph is constructed by choosing one adaptation action. A configuration can be either "intermediate" or "candidate." A candidate satisfies the allocation constraint that the sum of all VMs' CPU and memory capacities on each host must be less than 100%, while an intermediate does not satisfy the constraint. For instance, Mistral may assign more CPU capacity to a VM than available on a host by choosing an

"Increase CPU" action, requiring a subsequent "Reduce CPU" or "Migrate" action to yield a candidate. When a candidate c_i is determined to be the final optimal configuration, the shortest path starting from the initial configuration c to configuration c_i denotes the optimal sequence of adaptation actions needed to achieve optimal utility for a given control window.

Naive A* algorithm. Although the problem can be formulated as a weighted shortest path problem, it is not possible to fully explore the extremely large configuration space at runtime. To tackle this challenge without sacrificing optimality, we adopt the A* graph search algorithm [8]. The A* algorithm requires a "cost-to-go" heuristic to be associated with each vertex of the graph. This heuristic estimates the shortest distance from the vertex to a goal state and, for the result to be optimal, the heuristic must be "permissible" in that it overestimates the cost-to-go. We use the ideal utility U^* as the heuristic since it represents the highest utility that can be generated for the given workload. Since it does not consider any costs, it overestimates the utility and therefore, it is a permissible heuristic.

The algorithm starts from v_0 with current configuration. In each iteration of search, it generates the set of child vertices as one adaptation step from a parent vertex (e.g.,v_0) and stores these vertices in the total set of explored vertices V. It also stores the parent vertex only if it is a candidate configuration. It then chooses the vertex v from V with the lowest utility. Each vertex's utility is computed by summing the cost of actions from v_0 plus the cost-to-go if the vertex is an intermediate, or the total utility U if the vertex is a candidate. If the chosen vertex v is a candidate, the algorithm returns the vertex and computes actions. The algorithm considers v as the final optimal configuration since the vertex's utility is larger than any other utilities of intermediate configurations that can be generated by further explorations. This is because those utilities (i.e., the cost-to-go plus accumulated cost) will decrease as further actions are taken. Meanwhile, utilities of any other candidates generated by further explorations are less than those utilities generated with cost-to-go by the definition of permissible. Thus, optimality is guaranteed.

Since the naive A* algorithm still evaluates a large number of configurations due to the numerous possible adaptation actions at each depth of the graph, the search time may increase exponentially as the number of hosts and applications increases. For example, if the workload changes significantly and then stays relatively long in this state (resulting in a large control window), the algorithm may try to change the current configuration significantly by searching a large number of possible action sequences. The huge search space, and the resulting long search time, is a general problem for many optimization techniques proposed in the literature for cloud computing environments. Spending too much time to compute an optimal configuration can adversely affect the system response time (and utility) since the current configuration that may not be optimal for the changed workload is used during the decision making. Furthermore, the optimization procedure itself may consume significant amount of power while making its decision[2]. Therefore, we consider the cost of decision as another tradeoff in our optimization formulation.

Self-Aware A* algorithm. We have developed a method to accelerate the search by decreasing the search space at each vertex dynamically (i.e., decreasing the number of adaptation actions considered for each configuration in the graph). We set the algorithm to choose a small portion of all possible expanded configurations based on similarity to the ideal configuration c^*. Specifically, the algorithm computes a weighted Euclidean distances between each expanded configuration and c^* by summing up the differences in the corresponding VM sizes (CPU capacities) in the two configuration. We also set a weight to each VM based on its relative size in the ideal configuration. For example, we set 2 times more weight to VM_i than VM_j if their CPU capacities are 60% and 30%, respectively, in the ideal configuration. In addition, we compute another distance value based on VM placement on hosts by counting how many VMs have identical locations (same host) in the two configurations and then normalize the value with the total number of VMs.

Our Self-Aware A* algorithm uses the weighted Euclidian distances and a heuristic to dynamically restrict the search space and to allow Mistral to control the cost of search versus the potential benefits during the search process. Specifically, Mistral measures the elapsed time of the search, T, the utility accrued of the current configuration, U_T, and the power usage of the search procedure itself, U_{pwr^T}. Then, the algorithm compares the cost to an "expected utility", U_H, to decide when the search space needs to be restricted (i.e., search needs to be completed soon). We consider a history of recent utilities and choose the lowest one as U_H (i.e., a pessimistic estimate). Furthermore, we set a delay threshold for the search $\mathcal{T}$ that depends on the length of control window and can be empirically obtained. We use 5% of the control window length in our experiments. This threshold prevents a too long search in the case U_H is too high for the current system state. When the cost of search reaches U_H, or T exceeds $\mathcal{T}$, Mistral accelerates its search by decreasing search width of each vertex. The resulting search algorithm is shown in Algorithm 1. Note that the risk of stopping too early and never finding the correct adaptations is reduced by the fact that Mistral operates multi-level controllers and lower level controllers will refine the configuration chosen by the higher level controllers.

The algorithm takes the current configuration c, workload W, the length of control window CW, the expected utility U_H, its performance and power utilities over the unit monitoring interval U_{RT^H} and U_{pwr^H}, and the search delay threshold $\mathcal{T}$ as inputs and returns the optimal sequence of adaptation actions A. Using `Perf-Pwr`, the algorithm computes the ideal utilities. The `UtilityEst` estimates performance and power utilities, U'_{RT} and U'_{pwr}, with current configuration and workload. The elapsed time T, the utility accrued by the current configuration U_T, the power consumption incurred by

[2]"consuming power to save power"

Input: c, W, CW, U_H, U_{RT^H}, U_{pwr^H}, $\mathcal{T}$ **Output:** A

$(c^*, U_{RT^*}, U_{pwr^*}) \leftarrow$ `Perf-Pwr`(W);
if $c^* = c$ **then return** "*null*";
$v_0.(A, c, U_{RT}(A), U_{pwr}(A), U_{RT}, U_{pwr}, D)$
$\leftarrow (\phi, c, 0, 0, U_{RT^*}, U_{pwr^*}, 0)$;
$(V, T, U_T, U_{pwr^T}, st) \leftarrow (\{v_0\}, 0, 0, 0,$ `Time()`$)$;
$(U'_{RT}, U'_{pwr}) \leftarrow$ `UtilityEst` (c, W);
while *forever* **do**
  $v_{max} \leftarrow \text{argmax}_{v \in V} v.U$; $t \leftarrow 0$;
  if $v_{max}.a_{last} =$ "*null*" **then return** $v_{max}.A$;
  foreach $a \in \mathcal{A} \cup$ "*null*" **do**
    $v_n \leftarrow v_{max}$; $v_n.A \leftarrow v_{max}.A \cup a$;
    $v_n.c \leftarrow$ `NewConfig` $(v_n.c, a)$;
    $V_n \leftarrow V_n \cup v_n$;
  if $(U_T + U_{pwr^T}) \geq U_H$ *or* $(T \geq \mathcal{T})$ **then**
  $V_n \leftarrow$ `PruneByDistance`$_{v_n \in V_n}(v_n.c, c^*)$;
  foreach $v_n \in V_n$ **do**
    if $v_n.c =$ "*candidate*" **then**
      $(U_{RT}, U_{pwr}) \leftarrow$ `UtilityEst` $(v_n.c, W)$;
      $v_n.U \leftarrow (CW - v_n.D) \cdot (U_{RT} - U_{pwr}) +$
      $(v_n.U_{RT}(A) - v_n.U_{pwr}(A))$;
    else
      $(d, U_{RT}(a), U_{pwr}(a)) \leftarrow$`Cost` $(v_n.c, W, a)$;
      $v_n.U_{RT}(A) \leftarrow v_n.U_{RT}(A) + d \cdot U_{RT}(a)$;
      $v_n.U_{pwr}(A) \leftarrow v_n.U_{pwr}(A) + d \cdot U_{pwr}(a)$;
      $v_n.D \leftarrow v_n.D + d$;
      $v_n.U \leftarrow (CW - v_n.D) \cdot (U^*_{RT} - U^*_{pwr}) +$
      $(v_n.U_{RT}(A) - v_n.U_{pwr}(A))$;
    if $\exists v' \in V$ *s.t.* $v'.c = v_n.c$ **then**
      if $v_n.U > v'.U$ **then** $v' \leftarrow v_n$;
    else
      $V \leftarrow V \cup v_n$;
  $t \leftarrow$ `Time()` $- st$; $st \leftarrow$ `Time()`; $T \leftarrow T + t$;
  $U_{pwr^T} \leftarrow U_{pwr^T} + t \cdot U_{pwr^t}$;
  $U_T \leftarrow U_T + t \cdot (U'_{RT} - U'_{pwr})$;
  $U_H \leftarrow U_H - t \cdot (U_{RT^H} - U_{pwr^H})$;

Algorithm 1: Optimal adaptation search

the search procedure itself U_{pwr^T}, and expected utility U_H are updated after each depth of search.

In each iteration in the `while` loop, the open vertex with the highest utility is selected as v_{max}. If this vertex's configuration is a "candidate" (i.e., its last action is "null"), then the algorithm considers the configuration as the optimal one and returns actions leading to the configuration as described in the Naive A* algorithm. Otherwise, it explores further by triggering all possible actions including "null" (i.e., "do nothing"). `NewConfig` generates a new vertex (configuration) resulting from performing action a in the current vertex. If the cost of the search (i.e., $U_T + U_{pwr^T}$) exceeds the expected utility, or the elapsed time exceeds the given delay threshold, the algorithm prunes the number of new vertices using the Euclidean distances described above by calling `PruneByDistance`. The pruning selects the top 5% of the vertices. When a new configuration is a "candidate", the algorithm invokes `UtilityEst` to estimate performance and power utilities. Otherwise, it invokes `Cost` to compute the adaptation costs such as accrued performance and power utilities (i.e, $U_{RT}(A)$ and $U_{pwr}(A)$, respectively), and then computes the total utility with the cost-to-go values. If the newly generated vertex v_n is the same as one previously found, say v, and v_n's utility is larger than that of v, then the algorithm replaces the old vertex with the new one.

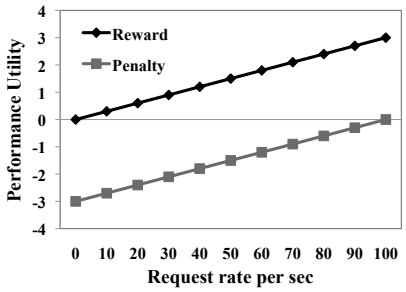


Fig. 3: Performance utility function

V. Experimental Evaluation

A. Experiment Setup

Our experimental testbed is illustrated by the test-bed box in Figure 2. We use a three-tier servlet version of the RUBiS benchmark [9] that emulates an eBay-like auction as our test applications. This application consists of an Apache web server, a Tomcat application server, and a MySQL database server running on a Linux-2.6 guest OS using Xen 3.2 [10]. In our experiments, we use "browsing only" transaction mix provided by the RUBiS package consisting of 9 read-only transaction types such as browse-category, browse-items, and view-comments. The application workload will remain in the range 0 to 100 req/sec. We use up to 4 applications, referred to as from RUBiS-1 to RUBiS-4 and thereby, deploy up to 20 VMs in the test-bed.

Hosts are commodity Pentium-4 1.8GHz machines with 1GB of memory running on a single 100Mbps Ethernet segment. Up to 8 machines are used to host RUBiS applications, while two are used as client emulators to generate workloads (not shown in the figure). One machine is dedicated to host dormant VMs used in server replication, and one as a storage server for VM disk images. Finally, we run the Mistral controller on a separate machine. We hook all machines except clients to a power meter to measure power usage. Each VM is allocated 200MB of memory with a limit of up to 4 VMs per host. The remaining 200MB is allocated to the Xen hypervisor, called Dom-0. The total CPU capacity of all VMs on a host is capped to 80% to ensure enough resources for Dom-0 even under loaded conditions. We set the minimum CPU capacity for each VM to 20% to avoid request errors even under low request rates. We use Xen's credit-based scheduler to dynamically set CPU capacity of each VM. To replicate the database server, we use a simple master-slave mechanism provided by MySQL. All tables are copied and synchronized between replicas. We set the maximum replication level for Tomcat and MySQL servers to 2, which is enough to handle the maximum request rates (100 req/sec) in our experiments, while we do not replicate Apache since a single Apache server per application is enough even under the maximum request rates.

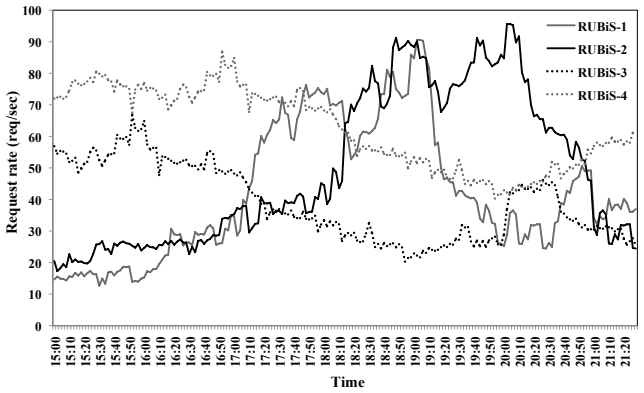


Fig. 4: Application workloads

We set the target response time to 400ms. This time was derived experimentally as the mean response time across all transactions of the RUBiS application running with a "default configuration" where all tiers' CPU capacities are set to 40% and workload was constant at 50 req/sec. The exact amount of the reward and penalty depends on the current application request rate and is shown in Figure 3. As the request rate increases, the reward increases and the penalty decreases to reflect an increasingly "best-effort" nature of service. In general, the rewards were chosen so as to yield a 20% "net profit" over the power costs incurred in the default configuration, and then scaled according to the workload. The monitoring interval is set to 2 minutes so that we can react quickly to workload changes. The cost per watt consumed over a monitoring interval was set to $ 0.01 in our experiments. We set the workload band to 0 req/sec for the 1^{st}-level controller and 8 req/sec for the 2^{nd}-level controller to ensure that even relatively small workload changes could cause the controller to be triggered.

In our experiments, we generate 4 workloads, one for each RUBiS application, based on the Web traces from the 1998 World Cup site [11] and the traffic traces of an HP customer's Internet Web server system [12]. We choose a typical day's traffic from each trace. Then we scale and shift them to the range of request rates that our experimental setup can handle. Specifically, we scale both the World Cup request rates of 150 to 1200 req/sec and the HP traffic of 2 to 4.5 req/sec to our desired range of 0 to 100 req/sec. Since our workload is controlled by adjusting the number of simulated clients, we create a mapping from the desired request rates to the number of simulated concurrent sessions. Figure 4 shows these scaled workloads for four RUBiS applications from 15:00 to 21:30, where RUBiS-1 and RUBiS-2 use the scaled World Cup trace, and RUBiS-3 and RUBiS-4 use the HP workload trace.

B. Model validation

We have validated the application performance models, the power models, and the stability interval estimation using the workloads in Figure 4. Figure 5 shows that our performance models (LQNM) provide sufficient accuracy for (a) response time and (b) utilization. The estimation error is approximately 5%. In this model validation, we use RUBiS-1 and RUBiS-2 with an interval of the workloads. As shown in Figure 4, the interval from 16:52 to 17:14 represents the first flash crowd in the scenario. While the Performance Manager generates a series of configurations using models for given request rates, we record estimated response times and CPU utilizations. Then we compare them with experiment results. In these experiments, we restart Mistral to measure values at each time point separately for each configuration and request rate to remove any noise caused by adaptations. Figure 5 (c) shows the model accuracy for power consumption measured using the same methodology as the response time and utilization accuracy validation.

Figure 6 illustrates the accuracy of the stability interval estimation when deploying RUBiS-1 and RUBiS-2. The average error is approximately 14%.

Finally, Figure 7 illustrates some of the adaptation costs measured for our applications on our testbed. The figures illustrate that adaptation costs are heavily influenced by the workload. We also measure power overhead and duration offline for shutting down/restarting hosts. Starting a host takes around 90 sec and consumes around 80 watts while shut-down takes 30 sec and consumes 20 watts. We assume that response times on other machines are not changed during these actions.

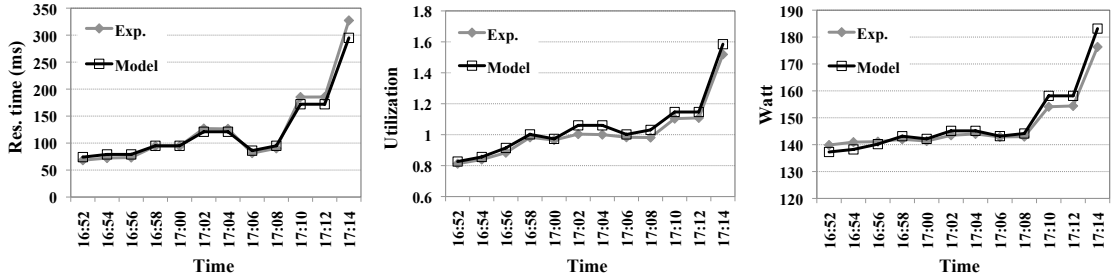
C. Adaptation Comparison

To evaluate our approach, we compare Mistral's results with those of three different approaches, each of which solves the tradeoff between two objectives among performance, power consumption, and adaptation costs as follows:

Perf-Pwr addresses the tradeoff between performance and power consumption, but ignores transient adaptation costs. We use our Perf-Pwr optimizer described in Section IV-A. In this approach, once a workload change is observed in a monitoring interval, the optimizer chooses adaptation actions and executes them.

Perf-Cost multiplexes a given fixed pool of resources to hosted applications to maximize performance utility. We allocate 2 hosts per application; this allocation can deal with the peak request rate in our scenario. This approach incorporates adaptation costs (adaptation duration and performance overhead) into the optimization formulation in each control window. However, it considers neither further power savings by consolidating VMs to a smaller number of hosts, nor power overhead during adaptations.

Pwr-Cost minimizes power consumption and adaptation costs under the restriction that VMs' CPU capacities for each request rate are given and static. These CPU capacities are large enough that the target response time can be met. To compute such VMs' capacities, we modify the Perf-Pwr optimizer so that it will not reduce the VM sizes below the capacity needed to meet the target response times. Given these VMs' capacities at each execution, the Pwr-Cost optimizer first adjusts the VMs' capacities of the VMs in the current configuration to match the new sizes. If the resulting VM CPU capacities violate the capacity constraints on some host (the sum of VM capacities on a host must be less than 100%), the VMs are



(a) Response times (b) Utilizations (c) Power consumptions

Fig. 5: Model accuracy

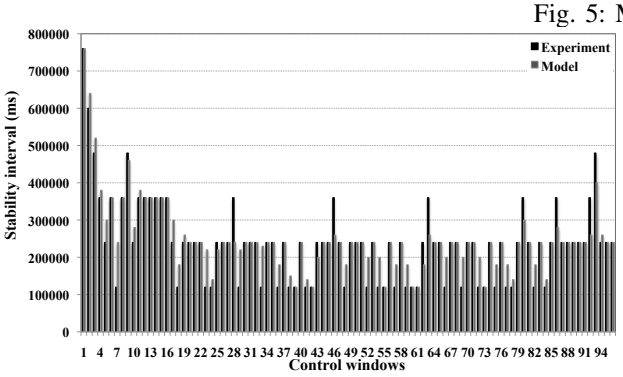


Fig. 6: Accuracy of stability interval estimation

migrated starting from the smallest one until the constraints are satisfied on all host. Finally, when no constraints are violated, the algorithm uses VM migration to consolidate VMs on fewer hosts if possible. During consolidation decision, it considers the tradeoff between power saving through consolidation and migration cost. Compared to Mistral, this approach never allows response time goals to be missed in order to reduce power usage or transient costs.

We compare these four approaches using RUBiS-1 and RUBiS-2 as described in Section V-A. Figures 8 through 9 show the results of the comparison. As demonstrated by Figure 8 (a) and (b), the response times with Mistral are somewhat higher than with other approaches, and it slightly violates the performance objective when request rates peak since it uses a maximum of 3 hosts out of the 4 to save power. However, due to more frequent and intensive adaptations in other approaches (shown as spikes in figures), performance violations with Perf-Pwr and Pwr-Cost are more frequent than with Mistral. Especially, the response times with Perf-Pwr fluctuate and then remain high since it performs many more adaptations than Mistral. Pwr-Cost is forced to execute migrations during the peak request rates to meet the capacity constraints since it does not trade off between performance and costs like Mistral.

Meanwhile, the overall power consumption with Mistral is lower than with the other approaches as illustrated in Figure 8 (c). This is because Mistral uses fewer hosts and performs fewer adaptations even under peak request rates. The curve of Perf-Pwr shows that using 4 hosts at peak request rates provides the optimal tradeoff between performance and power. However, Mistral chooses configurations with only 2 or 3 hosts since the cost of using 4 hosts would be too high. Pwr-Cost, however, is forced to use 4 hosts when both applications' request rates peak in order to host all the VMs with the required VM CPU capacities.

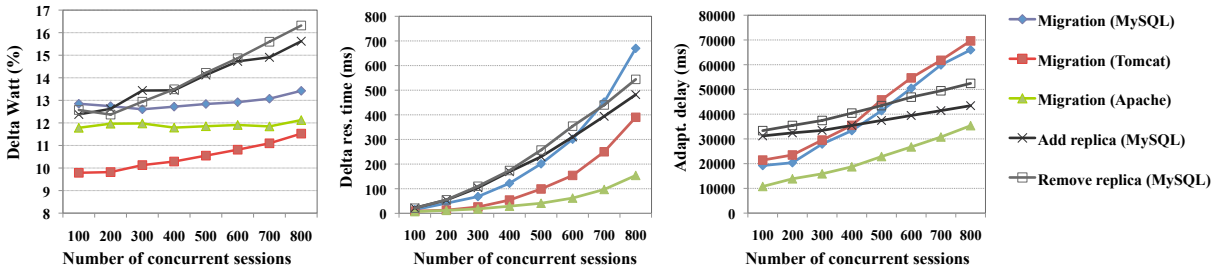
Finally, Figure 9 shows that the total utility of Mistral is indeed higher than the other approaches. For the duration of the experiment, the cumulative utility of Mistral (152.3) is higher than those of Perf-Pwr (-47.1), Perf-Cost (26.3), and Pwr-Cost (93.9). Although Perf-Cost provides a better response time behavior than Mistral, its utility is much lower than Mistral's since it consumes much more power. Thus, Mistral meets the goal of maximizing overall utility, consisting of performance and power utilities and transient costs, better than the other approaches.

D. Cost of Search

Next, we illustrate the cost of the decision making itself in terms of its power consumption, duration, and impact on the total utility. Specifically, we demonstrate that our Self-Aware search algorithm that is aware of its own execution costs can indeed result in significant improvement of overall utility. To measure the power consumption of Mistral's search algorithms, we connected only the host running Mistral to the power meter and then ran Mistral in a simulation mode where it only determines the action sequences to execute, but does not execute the adaptation actions chosen. Figure 10 (a) shows that the Mistral search algorithm consumes power up to 12 % over the host's idle power usage (i.e., 60 watts).

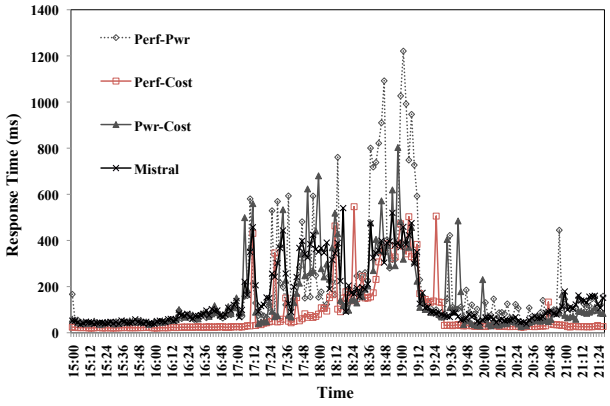
The next two experiments measured how the awareness of its own execution costs impacts the search algorithms. Figure 10(b) shows that the execution time of the naive search approach is up to 4 time longer (around 24 sec) than that of the Self-Aware search algorithm (around 5.5 sec) in the most intensive search cases. The longer search not only uses more power, but also keeps the system in the current configuration, which is not necessary close to optimal for the current workload, a longer time when the search for new configuration is in progress.

Finally, we show that such cost awareness does indeed improve the total utility. Figure 10 (c), based on a 2-application scenario (RUBiS-1 and RUBiS-2), shows that the utility of the naive approach is generally slightly lower than that of the Self-Aware approach, although the Naive approach typically executes more adaptation actions. The difference in the cumulative utilities over the execution time period is significant,

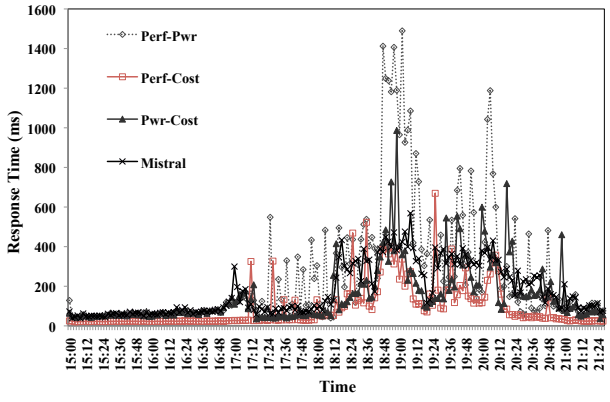


(a) Delta power consumption (b) Delta response times (c) Adaptation delay

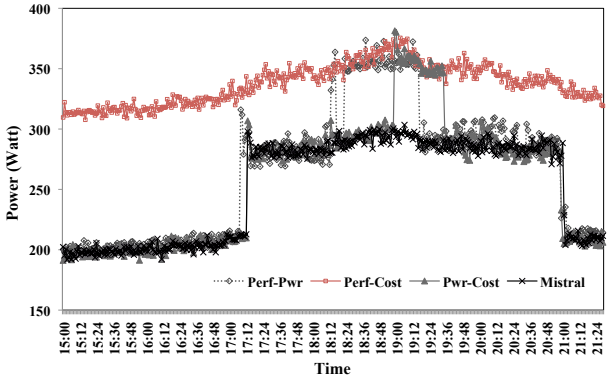
Fig. 7: Adaptation costs



(a) RUBiS-1 Response Time



(b) RUBiS-2 Response Time



(c) Power Consumption

Fig. 8: Comparison of Control Strategies

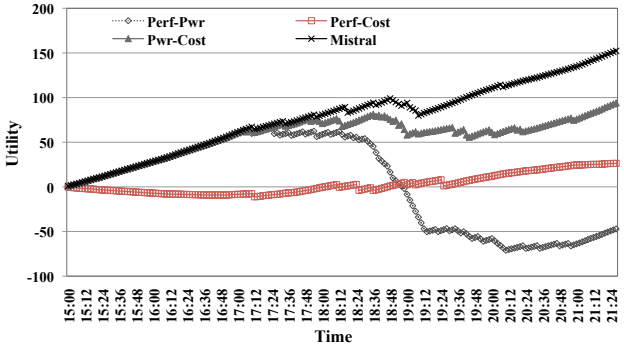
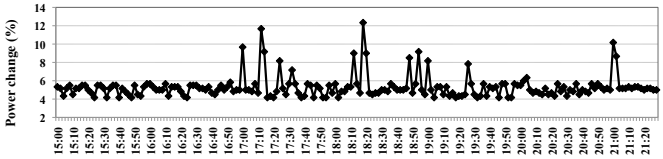
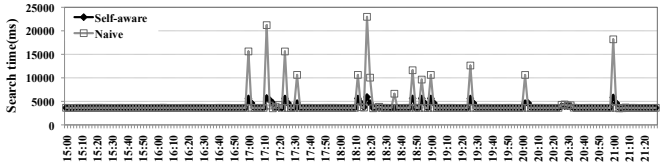


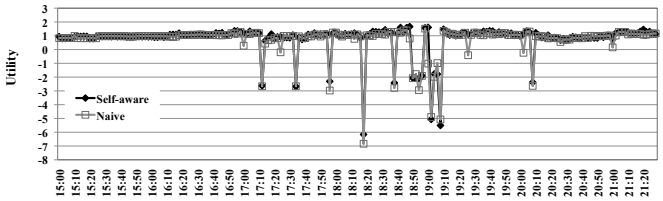
Fig. 9: Cumulative utility



(a) Power consumption



(b) Duration



(c) Utility

Fig. 10: Cost of Search

with cumulated utilities of 135.3 (Naive) and 152.3 (Self-Aware).

E. Scalability

Finally, we demonstrate how Mistral scales to larger numbers of applications and hosts and discuss its use in managing large-scale data centers. We deploy up to 20 VMs of 4 RUBiS applications to all given hosts (i.e., 8 hosts). For the 3-app scenario, we add RUBiS-3 to the scenario used in the above

TABLE I: Search durations and utilities

	2-app	**3-app**	**4-app**
#VMs / #hosts	10 / 4	15 / 6	20 / 8
Self-Aware (avg. duration)	3807.8	5669.9	7514.8
- 1^{st} level	3737.6	4977.2	5956.8
- 2^{nd} level	5287.4	8029.7	10797.4
Naive (avg. duration)	4341.4	11343.4	35155.8
- 1^{st} level	4077.5	5798.7	11615.9
- 2^{nd} level	13387.2	59345.6	250297.4
Mistral (total utility)	152.3	336.6	504.8
Ideal (total utility)	351.7	538.3	701.9

experiments and then for the 4-app scenario, we add RUBiS-4. We configure Mistral to use two-level controller. Each 1^{st} level controllers operates with workload band of 0, manages a subset of the hosts, and uses CPU tuning and VM migrations within its managed subset. The 2^{nd} level controller operates with a workload band of 8 req/sec, controls the whole system, and uses all the actions introduced in this paper. The 2-app scenario uses one controller on each level (with the 1^{st} level one controlling all 4 machines), while the 3-app and 4-app scenarios use 2 1^{st} level controllers and one 2^{nd} level controller.

Table I summarizes results of 3 different scenarios. We report the average search times for the Naive and the Self-Aware controllers as well as the averages for each level's controllers. As the number of hosts and applications increase, the search space of adaptation actions to consider increases exponentially. The search duration of the Naive search algorithm illustrates the exponential increase. To tackle this problem, our Self-Aware search algorithm restricts the search space when necessary using simple technique based on weighted Euclidean distances. The results show that the duration for Self-Aware algorithm increases approximately linearly with the number of machines, while generating reasonable utilities. To estimate the optimality of our approach, we compare these utilities to the ideal utilities generated by the simulated Perf-Pwr optimizer that ignores adaptation costs. The results show that the gap between the achieved and ideal utilities in each scenario remains approximately constant.

These results have implications on using Mistral to manage an entire data center or a cloud platform. Centralized optimization techniques are typically not scalable enough to manage a large system consisting of 100s of machines in real-time (e.g., execute every few minutes). Mistral can address this challenge due to its ability to implement multi-level hierarchical control. Specifically, local controllers managing a few machines can execute relatively frequently (every few minutes), while higher-level controllers can operate hourly or daily on larger groups of machines, e.g., a rack or the whole data center. We plan to integrate more adaptation actions that need larger time-scale management, such as migration over WAN and disk image migration between data centers, into our framework.

VI. Related Work

We categorize related work based on adaptation methodologies and objectives. Many efforts have tackled intelligent power control using underlying hardware support such as processor throttling and low-power DRAM states. In particular, Dynamic voltage and frequency scaling of processors (DVFS) has been adopted by many authors including [13], [14], [15], [16], but mainly in single-server settings. Extending these control algorithms to balance end-to-end performance across multiple tiers and clusters remains a challenge. Nevertheless, we believe that techniques such as DVFS are complementary to ours and can be incorporated into the lowest level controllers in our approach.

Some researchers have worked to maximize the use of a given power budget across multiple machines and tiers. In [3], the authors present a methodology for efficient power provisioning that increases the number of services that can be deployed within a given power budget. Govindan et al. have tackled a similar problem using statistical multiplexing methods to improve the power utilization in [17]. However, they do not explicitly consider the power-performance tradeoff nor any transient costs.

A number of projects have addressed different aspects of the power-performance tradeoff. Gandhi et al. use queuing models to find the optimal power allocation among servers so as to minimize mean response time under a given power budget in [18], while Kephart et al. address the tradeoff in [19] using reinforcement learning over a decentralized architecture in which power and performance managers cooperate. Chase et al. discuss turning servers on and off for efficient power management in [20]. However, these approaches have not considered adaptation costs which, as we have shown, can have a significant impact on overall utility. Similar to our approach, Chen et al. consider some adaptation costs (time overheads and wear-and-tear) [14], but they do not consider process migration and consolidation.

The authors of [21] propose a technique that exploits the hypervisor's ability to limit hardware usage of VMs and control power consumption of individual VMs in a fine-grained manner. The mechanism can be integrated with our approach as an adaptation action to achieve further power savings at an aggregated level.

Recently, a number of power management systems have been proposed based on virtualization techniques, including [22], [23], [5], [24], [6], that share some adaptation methods with our approach. Tolia et al. demonstrate the ability of such techniques to optimize the performance-power tradeoff in two case studies using COTS hardware [23]. Cardosa et al. control min, max, and share parameters of VMs to manage the power-performance tradeoff and develop constrained bin-packing algorithms [24]. However, they do not consider benefits and costs of VM migrations that can be used to further consolidate servers by packing VMs into a smaller number of physical machines in such virtualized environments.

Kusic et al. tackles a similar problem of achieving power

efficiency while maintaining the desired performance by consolidating servers, and also explicitly deals with transient costs [22]. While they consider the potential excessive costs caused by high workload variations in their problem formulation, they only consider a single type of adaptation (turning on/off machines). Moreover, adding multiple actions to their approach is not trivial and can lead to significant challenges with scalability.

The pMapper system [5] tackles power-cost tradeoffs under a fixed performance constraint by using modified bin-packing algorithms to minimize migration costs while packing VMs in a small number of machines. Our Pwr-Cost approach is inspired by pMapper. Similarly, Sanjay et al. perform VM placement to save power without degrading performance [6]. They also consider adaptation costs to improve system stability in their distributed architecture. However, their focus is on developing an extensible architecture to coordinate various management objectives, rather than solving tradeoffs between those objectives.

VII. Conclusions

Managing large computer systems (e.g., data centers, clouds) with complex multitier distributed applications is becoming increasing important and challenging due to often conflicting goals of meeting performance objectives, saving power, and managing the cost of management decisions and actions. In this paper, we have presented Mistral, a control architecture that optimizes total utility that includes application utility due to meeting/missing performance objectives, power costs, and transient adaptation costs. We demonstrate experimentally that Mistral provides better overall utility than a number of alternative controllers that consider only a subset of these factors. To our knowledge, our self-aware search algorithm is the first one to consider the cost of the search itself in its decision making. We demonstrate experimentally that such self-awareness does indeed improve overall total utility. Mistral can also be configured as a multi-level hierarchical controller enabling its potential application in large scale systems.

References

[1] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proc. USENIX Sym. on Networked Systems Design and Implementation*, 2005.

[2] G. Jung, K. Joshi, M. Hiltunen, R. Schlichting, and C. Pu, "A cost-sensitive adaptation engine for server consolidation of multitier applications," in *Proc. ACM/IFIP/USENIX Int. Middleware Conf.*, 2009.

[3] X. Fan, W. Weber, and L. Barroso, "Power provisioning for a warehouse-sized computer," in *Proc. of the* 34^{th} *ACM Int. Sym. on Computer Architecture*, 2007, pp. 13–23.

[4] G. Jung, K. Joshi, M. Hiltunen, R. Schlichting, and C. Pu, "Generating adaptation policies for multi-tier applications in consolidated server environments," in *Proc. IEEE Int. Conf. on Autonomic Computing*, 2008, pp. 23–32.

[5] A. Verma, P. Ahuja, and A. Neogi, "pmapper: Power and migration cost aware application placement in virtualized systems," in *Proc. of the* 9^{th} *ACM/IFIP/USENIX International Middleware Conference*, 2008, pp. 243–264.

[6] S. Kumar, V. Talwar, V. Kumar, P. Ranganathan, and K. Schwan, "vmanage: Loosely coupled platform and virtualization management in data centers," in *Proc. IEEE Int. Conf. on Autonomic Computing*, 2009, pp. 127–136.

[7] G. Box, G. Jenkins, and G. Reinsel, *Time Series Analysis: Forecasting and Control*, 3rd ed. Upper Saddle River, New Jersey: Prentice Hall, 1994.

[8] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

[9] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel, "Performance comparison of middleware architectures for generating dynamic web content," in *Proc. of the* 4^{th} *ACM/IFIP/USENIX Int. Middleware Conf.*, 2003.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Wareld, "Xen and the art of virtualization," in *Proc. of 19th ACM Sym. on Operating Systems Principles*, 2003, pp. 164–177.

[11] M. Arlitt and T. Jin, "Workload charaterization of the 1998 world cup web site," in *HP Technical Report*, 1999.

[12] J. Dilley, "Web server workload charaterization," in *HP Technical Report, HPL-96-160*, 1996.

[13] T. Pering, T. Burd, and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algortithms," in *Int. Sym. on Low Power Electronics and Design*, 1998.

[14] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam, "Managing server energy and operational costs in hosting centers," in *Proc. of the 2005 ACM SIGMETRICS*, 2005, pp. 303–314.

[15] W. Felter, K. Rajamani, C. Rusu, and T. Keller, "A performance-conserving approach for reducing peak power consumption in server systems," in *Proc. of the* 19^{th} *Annual Int.l Conf. on Supercomputing*, 2005, pp. 293–302.

[16] C. Lefurgy, X. Wang, and M. Ware, "Power capping: A prelude to power shifting," *Cluster Computing*, vol. 11, no. 2, pp. 183–195, May 2008.

[17] S. Govindan, J. Choi, B. Urgaonkar, A. Sivasubramaniam, and A. Baldini, "Statistical profiling-based techniques for effective power provisioning in data centers," in *ACM European Conf. on Computer Systems*, 2009, pp. 317–330.

[18] A. Gandhi, M. Harchol-Balter, R. Das, and C. Lefurgy, "Optimal power allocation in server farms," in *Proc. of the 2009 ACM SIGMETRICS*, 2009.

[19] J. Kephart, H. Chan, R. Das, D. Levine, G. Tesauro, F. Rawson, and C. Lefurgy, "Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs," in *Int. Conf. on Autonomic Computing*, 2007, pp. 24–33.

[20] J. Chase, D. Anderson, and P. Thakar, "Managing energy and server resources in hosting centers," in *Proc. of the* 18^{th} *Sym. on Operating Systems Principles*, 2001, pp. 103–116.

[21] R. Nathuji and K. Schwan, "Virtualpower: Coordinated power management in virtualized enterprise systems," in *Proc. of the* 21^{st} *ACM SIGOPS Symposium on Operating Systems Principles*, 2007, p. 265 278.

[22] D. Kusic, J. Kephart, J. Hanson, N. Kandasamy, and G. Jiang, "Power and performance management of virtualized computing environments via lookahead control," in *Int. Conf. on Autonomic Computing*, 2008, pp. 3–12.

[23] N. Tolia, Z. Wang, M. Marwah, and C. Bash, "Delivering energy proportionality with non energy-proportional systems - optimizing the ensemble," in *Proc. of the* 1^{st} *USENIX Workshop on Power Aware Computing and Systems*, 2008.

[24] M. Cardosa, M. Korupolu, and A. Singh, "Shares and utilities based power consolidation in virtualized server environments," in *Proc. of the* 11^{th} *IFIP/IEEE International Symposium on Integrated Network Management*, 2009.