Enabling resource sharing between transactional and batch workloads using dynamic application placement

David Carrera[1], Malgorzata Steinder[2], Ian Whalley[2], Jordi Torres[1], and Eduard Ayguadé[1]

[1] Technical University of Catalonia (UPC) - Barcelona Supercomputing Center (BSC)
Barcelona, Spain
{david.carrera, jordi.torres, eduard.ayguade}@bsc.es
[2] IBM T.J. Watson Research Center
Hawthorne. NY 10532
{steinder, inw}@us.ibm.com

Abstract. We present a technique that enables existing middleware to fairly manage mixed workloads: batch jobs and transactional applications. The technique leverages a generic application placement controller, which dynamically allocates compute resources to application instances. The controller works towards a fairness goal while also trying to maximize individual workload performance. We use relative performance functions to drive the application placement controller. Such functions are derived from workload-specific performance models—in the case of transactional workloads, we use queuing theory to build the performance model. For batch workloads, we evaluate a candidate placement by calculating long-term estimates of the completion times that are achievable with that placement according to a scheduling policy. In this paper, we propose a lowest relative performancwe first scheduling policy as a way to also achieve fair resource allocation among batch jobs. Our technique permits collocation of the workload types on the same physical hardware, and leverages control mechanisms such as suspension and migration to perform online system reconfiguration. In our experiments we demonstrate that our technique maximizes mixed workload performance while providing service differentiation based on high-level performance goals.

1 Introduction

Transactional applications and batch jobs are widely used by many organizations to deliver critical services to their customers and partners. For example, in financial institutions, transactional web workloads are used to trade stocks and query indices, while computationally intensive non-interactive workloads are used to analyse portfolios or model stock performance. Due to intrinsic differences among these workloads, they are typically run today on separate dedicated hardware, which contributes to resource under-utilization and management complexity. Therefore, organizations demand management solutions that permit such workloads to run together on the same hardware, improving resource utilization while continuing to offer performance guarantees.

Integrated performance management of mixed workloads is a challenging problem. First, performance goals for different workloads tend to be of different types. For interactive workloads, goals are typically defined in terms of average or percentile response time or throughput over a short time interval, while goals for non-interactive workloads concern the performance (e.g., completion time) of individual jobs. Second, due to the nature of their goals and short duration of individual requests, interactive workloads lend themselves to automation at short control cycles, whereas non-interactive workloads typically require calculation of a schedule for an extended period of time.

In addition, different types of workload require different control mechanisms for management. Transactional workloads are managed using flow control, load balancing, and application placement. Non-interactive workloads need scheduling and resource control. Traditionally, these have been addressed separately.

To illustrate the problems inherent in managing these two types of workload together, let us consider a simple example. Consider a system consisting of 4 identical machines. At some point in time, in the system there is one transactional application, TA, which requires the capacity of 2 machines to meet its average response time goal. The system also includes 4 identical batch jobs, each requiring one physical machine for a period of time t and having completion time goal of $T = 3t$. The jobs are placed in a queue and are labeled J1, J2, J3, and J4, according to their order in the queue. The system must decide how many jobs should be running—that is, how many machines should be allocated to the transactional application and to batch jobs respectively. Let us consider two of the possible configurations. In the first configuration, one machine is allocated to batch workload and three machines are used by TA. Thus, jobs execute in sequence and complete after time t, $2t$, $3t$, and $4t$. As a result, J4 violates its SLA goal, while TA overachieves its SLA target. In the second configuration, two machines are allocated to batch workload, which permits the four jobs to complete at times t, t, $2t$ and $2t$, respectively. Thus all jobs exceed their SLA goal, while TA also meets its SLA target. Clearly, the second configuration is a better choice.

Let us now assume that that the second configuration is put into effect, but then, at time $t/2$, the workload intensity for TA increases such that it now requires all 4 machines to meet its SLA goal. In the current configuration, all jobs will exceed their SLA goals, but TA will violates its goal. If, for the sake of easy calculation, we assume that the response time of TA is proportional to the inverse of its allocated capacity, then TA will violate its response time goal by 100%. Therefore, it makes sense to consider suspending one of the running jobs, J2, and allocating its capacity to TA. If this change occurs at time $t/2$, then J1, J2, J3, and J4, complete at times t, $1.5t$, $2.5t$, and $3.5t$ respectively—all jobs run in series on a single machine, and J2 resumes halfway through its execution. Thus, J1, J2, and J3 exceed their SLA goals, J4 violates its goal by about 16%, and TA violates its goal by about 33%. This results in an allocation that, when the goals of all workloads cannot be met, spreads goal violations among workloads so as to achieve the smallest possible violation for each application.

These examples show that in order to manage resource allocation to a mix of transactional and batch workloads, the system must be able to make placement decisions at short time intervals, so as to respond to changes in transactional workload intensity. While making decisions, the system must be able to look ahead in the queue of jobs and

predict the future performance (relative to goals) of all jobs—both those started now, and those that will be started in the future. It must be able to make trade-offs between the various jobs and the transactional workload, taking into account their goals.

Enabling resource sharing between transactional and batch workloads also introduces a number of challenges in the area of application deployment, update, configuration, and performance and availability management. Many of these challenges are addressed by virtualization technologies, which provide a layer of separation between a hardware infrastructure and workload, and provide a uniform set of control mechanisms for managing these workloads embedded inside virtual containers. Our technique relies on common virtualization control mechanisms to manage workloads.

In addition, our system uses Relative Performance Functions (RPF from here on) to permit trade-offs between different workloads. The RPFs define application performance relative to that application's goal. It can therefore be seen that equalizing the achieved relative performance between two applications results in "fairness"—the applications will be equally satisfied in terms of relative distance from their goals. The original contribution of this paper is a scheme for modeling the performance of, and managing, non-interactive long-running workloads.

This paper is organized as follows. In Section 2, we explain the contributions of this paper in the context of related work. In Section 3, we present our approach to managing heterogeneous workloads using resource allocation driven by application relative performance. In Section 4, we describe the calculation of the relative performance function for non-interactive applications. In Section 5 we evaluate our approach via simulation.

2 Related work

The explicit management of heterogeneous workloads was previously studied in [1], in which CPU shares are manually allocated to run mixed workloads on a large multiprocessor system. This is a static approach, and does not run workloads within virtual machines. Virtuoso [2] describes an OS scheduling technique, VSched, for heterogeneous workload VMs. VSched enforces compute rate and interactivity goals for both non-interactive and interactive workloads (including web workloads), and provides soft real-time guarantees for VMs hosted on a single physical machine. VSched could be used as a component of our system for providing resource-control automation mechanisms within a machine, but our approach addresses resource allocation for heterogeneous workloads across a cluster of machines.

The relative performance functions we use in our system are similar in concept to the utility functions that have been used in real-time work schedulers to represent the fact that the value produced by such a system when a unit of work is completed can be represented in more detail than a simple binary value indicating whether the work met its or missed its goal. In [3], the completion time of a work unit is assigned a value to the system that can be represented as a function of time. Other work in the field of utility-driven management are summarized in [4] with special focus on real-time embedded systems. In [5], the authors present a utility-driven scheduling mechanism that aims to maximize the aggregated system utility. Our technique does not focus on real-time systems, but on any general system for which performance goals can be expressed

as relative performance functions. In addition, we introduce the notion of fairness into our application-centric management technique—our objective is not to maximize the system relative performance, but to at least maximize the performance of the least performing application.

Outside of the realm of the real-time systems, the authors of [6] focus on a utility-guided scheduling mechanism driven by data management criteria, since this is the main concern for many data-intensive HPC scientific applications. In our work we focus on CPU-bound heterogeneous environments, but our technique could be extended to observe data management criteria by expanding the semantics of our RPFs.

Despite the similarity between an RPF and a utility function, one difference should be pointed out. While utility functions are typically used to model user satisfaction or business value resulting from a particular level of performance, an RPF is merely a measure of relative performance distance from the goal. Hence, unlike in [7, 8] we do not study the correctness of RPFs with respect to modeling user satisfaction. If such a satisfaction model exists, it may be used to transform an RPF into a utility function.

There is also previous work in the area of managing workloads in virtual machines. Management of clusters of virtual machines is addressed in [9] and [10]. The authors of [9] address the problem of deploying a cluster of virtual machines with given resource configurations across a set of physical machines. The authors of [10] define a Java VM API that permits a developer to set resource allocation policies. In [11] and [12], a two-level control loop is proposed to make resource allocation decisions within a physical machine, but these do not address integrated management of multiple physical machines. The authors of [13] study the overhead of a dynamic allocation scheme that relies on virtualization as opposed to static resource allocation. Their evaluation covers both CPU-intensive jobs and transactional workloads, but does not consider mixed environments. Neither of these techniques provides a technology to dynamically adjust allocation based on SLA objectives in the face of resource contention.

Placement problems in general have also been studied in the literature, frequently using techniques including bin packing, multiple knapsack problems, and multi-dimensional knapsack problems [14]. The optimization problem that we consider presents a non-linear optimization objective while previous approaches [15, 16] to similar problems address only linear optimization objectives. In [17], the authors evaluate a similar problem to that addressed in our work (restricted to transactional applications), and use a simulated annealing optimization algorithm. Their strategy aims to maximize the overall system utility while we focus on first maximizing the performance of the least performing application in the system, which increases fairness and prevents starvation, as was shown in [18]. In [19], a fuzzy logic controller is implemented to make dynamic resource management decisions. This approach is not application-centric—it focuses on global throughput—and considers only transactional applications. The algorithm proposed in [20] allows applications to share physical machines, but does not change the number of instances of an application, does not minimize placement changes, and considers a single bottleneck resource.

3 Integrated management of heterogeneous workloads

3.1 System architecture

We consider a system that includes a set of heterogeneous physical machines, referred to henceforth as *nodes*. Transactional web applications, which are served by application servers, are replicated across nodes to form *application server clusters*. Requests to these applications arrive at an entry router which may be an L4 or L7 gateway that distributes requests to clustered applications according to a load balancing mechanism. Long-running jobs are submitted to the *job scheduler*, placed in its queue, and dispatched based on the resource allocation decisions of the management system.

The request router monitors incoming and outgoing requests and measures their service times and arrival rates per application. It may also employ an overload protection mechanism [21, 22] by queuing requests that cannot be immediately accommodated by server nodes. A separate component, called the *work profiler* [23], monitors resource utilization of nodes and (based on a regression model that combines the utilization values with throughput data) estimates an average CPU requirement of a single request to any application. Based on these findings, our system builds performance models that allow it to predict the performance of any transactional application for any given allocation of CPU power. The size and placement of application clusters is determined by the *application placement controller* (APC).

Batch jobs are submitted to the system via the *job scheduler*. Each job has an associated performance goal. Currently we support completion time goals, and we plan to extend the system to handle other performance objectives. The job scheduler uses APC as an advisor as to where and when a job should be executed. When APC makes a decision, actions pertaining to batch jobs are given to the scheduler to be put into effect. The job scheduler also monitors job status and notifies APC, which uses the information in subsequent control cycles. A *job workload profiler* estimates job resource usage profiles, which are fed into APC. Job usage profiles are used to derive an RPF of a given resource allocation to jobs, which is used by APC to make allocation decisions.

APC operates in a control loop with period T, which is of the order of minutes. A short control cycle is necessary to allow the system to react quickly to transactional workload intensity changes which may happen frequently and unexpectedly. In each cycle, APC examines the placement of applications on nodes and their resource allocations, evaluates the relative performance of this allocation and makes changes to the allocation by starting, stopping, suspending, resuming, relocating or changing CPU share configuration of some applications. In the following sections we will concentrate on the problem solved by APC in a single control cycle.

3.2 Problem statement

We are given a set of nodes, $\mathcal{N} = \{1, \ldots, N\}$ and a set of applications $\mathcal{M} = \{1, \ldots, M\}$. We use n and m to index into the sets of nodes and applications respectively. With each node n we associate its memory and CPU capacities. With each application, we associate its load independent demand, that represents the amount of memory consumed by this application whenever it is started on a node. We use symbol P to denote a placement matrix of applications on nodes. Cell $P_{m,n}$ represents the number of instances of

application m on node n. We use symbol L to represent a load placement matrix. Cell $L_{m,n}$ denotes the amount of CPU speed consumed by all instances of application m on node n. A RPF for each application may be expressed as a function of L.

Use of Relative Performance Functions A relative performance function for a given application is a measure of the relative distance of the application's performance from its goal. It has a value of 0 when the application exactly meets its performance goal. Values greater than 0 and less than 0 represent the degree with which the goal is exceeded or violated, respectively. In our system we associate an RPF to each existing application.

RPFs are used to model the relation between delivered service level and application satisfaction. For resource allocation purposes, such functions can be transformed to model application satisfaction given particular resource allocation. Notice that application satisfaction can be understood as a measurement of relative performance. Section 3.3 and Section 4 describe how RPFs are calculated for both transactional applications and long-running jobs in our system. Our system aims to make fair placement decisions in terms of relative performance – application performance relative to its goal. The use of RPFs in our system is justified by the fact that they provide uniform workload-specific performance models that allow fair placement decisions across different workloads.

Although in our system we use linear functions, other models could be decided as it is discussed in section 2. Deciding the best shape for application performance models is out of the scope of this particular work, but the technique here presented will continue to work for any existing monotonic growing model.

Optimization objective Given an application placement matrix P and a load distribution matrix L, a relative performance value can be calculated for each application. The performance of the system can then be measured as an ordered vector of application relative performance. The objective of APC is to find the best possible new placement of applications as well as a corresponding load distribution such that maximizes the performance of the system.

The optimization objective is an extension of a $\max\min$ criterion, and differs from it by explicitly stating that after the $\max\min$ objective can no longer be improved (because the lowest performing application cannot be allocated any more resources), the system should continue improving the relative performance of other applications. The APC finds a placement that meets the above objective while ensuring that neither the memory nor CPU capacity of any node is overloaded. In addition, APC employs heuristics that aim to minimize the number of changes to the current placement. While finding the optimal placement, APC also observes a number of constraints, such as resource constraints, collocation constraints and application pinning, amongst others.

Algorithm outline The application placement problem is known to be NP-hard and heuristics must be used to solve it. In this paper, we leverage an algorithm proposed in [18].

The core of the algorithm is a set of three nested loops. An outer loop iterates over nodes. For each visited node, an intermediate loop iterates over application instances placed on this node and attempts to remove them one by one, thus generating a set of

configurations whose cardinality is linear in the number of instances placed on the node. For each such configuration, an inner loop iterates over all applications, attempting to place new instances on the node as permitted by the constraints.

The order in which nodes, instances, and applications are visited is driven by relative performance functions. In the process, the algorithm examines application relative performance asking the following questions:

- What is the relative performance of an application in the specified placement?
- Given application placement, how much additional CPU power must be allocated to an application such that it achieves the specified relative performance value?

In Section 3.3, we briefly explain how these questions are answered for web workloads. Section 4 introduces the relative performance function for long-running workloads, which is an original contribution of this paper.

3.3 Performance model for transactional workloads

In our system, a user can associate a response time goal, τ_m with each transactional application. Based on the observed response time for an application t_m, we evaluate the system performance with respect to the goal using an objective function u_m, which is defined as follows:

$$u_m(t_m) = \frac{\tau_m - t_m}{\tau_m} \tag{1}$$

We leverage the request router's performance model and the application resource usage profile to estimate t_m as a function of the CPU speed allocated to the application, $t_m(\omega_m)$. This allows us to express u_m as a function of ω_m, $u_m(\omega_m) = u_m(t_m(\omega_m))$.

Given a placement P and the corresponding load distribution L, we obtain $u_m(L)$ by taking $u_m(\omega_m)$, where $\omega_m = \sum_n L_{m,n}$. Likewise, we can calculate the amount of CPU power needed to achieve a relative performance u by taking the inverse function of u_m, $\omega_m(u)$.

The performance model for transactional workloads is not an original contribution of this work, but is in the core of the middleware upon which our work relies. Thus, the reader is referred to [21] for a detailed description of the model.

4 Performance model for non-interactive workloads

In this section, we focus on applying our placement technique to manage long-running jobs. We start by observing that a performance management system cannot treat batch jobs as individual management entities, as their completion times are not independent. For example, if jobs that are currently running complete sooner, this permits jobs currently in the queue (not yet running) to complete sooner as well. Thus, performance predictions for long-running jobs must be done in relation to other long-running jobs.

Another challenge is to provide performance predictions with respect to job completion time on a control cycle which may be much lower than job execution time. Typically, such a prediction would require the calculation of an optimal schedule for the jobs. To trade-off resources among transactional and long-running workloads, we would have to evaluate a number of such schedules calculated over a number of possible

divisions of resources among the two kinds of workloads. The number of combinations would be exponential in the number of nodes in the cluster. We therefore propose an approximate technique, which is presented in this section.

4.1 Job characteristics

We are given a set of jobs. With each job m we associate the following information:

Resource usage profile. A resource usage profile describes the resource requirements of a job and is given at job submission time—in the real system, this profile comes from the job workload profiler. The profile is estimated based on historical data analysis. Each job m consists of a sequence of N_m stages, $s_1, \ldots, s_{N_m}$, where each stage s_k is described by the following parameters:

- The amount of CPU cycles consumed in this stage, $\alpha_{k,m}$.
- The maximum speed with which the stage may run, $\omega_{k,m}^{\max}$.
- The minimum speed with which the stage must run, whenever it runs, $\omega_{k,m}^{\min}$.
- The memory requirement $\gamma_{k,m}$.

Performance objectives. The SLA objective for a job is expressed in terms of its desired completion time, τ_m, which is the time by which the job must complete. Clearly, τ_m should be greater than the job's desired start time, τ_m^{start}, which itself is greater than or equal to the time when the job was submitted. The difference between the completion time goal and the desired start time, $\tau_m - \tau_m^{\text{start}}$, is called the relative goal.

We are also given an RPF that maps actual job completion time t_m to a measure of satisfaction from achieving it, $u_m(t_m)$. If job m completes at time t_m, then the relative distance of its completion time from the goal is expressed by the RPF of the following form.

$$u_m(t_m) = \frac{\tau_m - t_m}{\tau_m - \tau_m^{\text{start}}} \tag{2}$$

Runtime state. At runtime, we monitor and estimate the following properties for each job: Current status, which may be either running, not-started, suspended, or paused; and CPU time consumed thus far, α_m^*.

4.2 Hypothetical relative performance

To calculate job placement, we need to define an RPF which APC can use to evaluate its placement decisions. While the actual relative performance achieved by a job can only be calculated at completion time, the algorithm needs a mechanism to predict (at each control cycle) the relative performance that each job in the system will achieve given a particular allocation. This is also the case for jobs that are not yet started, for which the expected completion time is still undefined. To help answer questions that APC is asking of the RPF for each application, we introduce the concept of *hypothetical relative performance*.

Estimating application relative performance given aggregate CPU allocation Suppose that we deal with a system in which all jobs can be placed simultaneously, and in which the available CPU power may be arbitrarily finely allocated among the jobs. We require a function that maps the system's CPU power to the relative performance achievable by jobs when placed on it.

Let us consider job m. Based on its properties, we can estimate the completion time needed to achieve relative performance u, $t_m(u) = \tau_m - u(\tau_m - \tau_m^{\text{start}})$. Then we can calculate the average speed with which the job must proceed over its remaining lifetime to achieve u, as follows:

$$\omega_m(u) = \frac{\alpha^{cr}_{N_m,m}}{t_m(u) - t_{\text{now}}} \tag{3}$$

where we define $\alpha^{cr}_{N_m,m}$ as the remaining work to complete all stages up to stage N_m, m.

To achieve the relative performance of u for all jobs, the aggregate allocation to all jobs must be $\omega_g = \sum_m \omega_m(u)$. To create the RPF, we sample $\omega_m(u)$ for various values of u and interpolate values between the sampling points.

Let $u_1 = -\infty, u_2, \ldots, u_R = 1$, where R is a small constant, be a set of sampling points (target relative performance values from now on). We define matrices W and V as follows:

$$W_{i,m} = \begin{cases} \omega_m(u_i) & \text{if } u_i < u_m^{\max} \\ \omega_m(u_m^{\max}) & \text{otherwise} \end{cases} \tag{4}$$

$$V_{i,m} = \begin{cases} u_i & \text{if } u_i < u_m^{\max} \\ u_m^{\max} & \text{otherwise} \end{cases} \tag{5}$$

Cell $W_{i,m}$ contains the average speed with which application m should execute starting from t_{now} to achieve relative performance u_i, and cell and $V_{i,m}$ contains the relative performance value u_i if it is possible for application m to achieve this performance level u_i. If relative performance u_i is not achievable by application m, these cells instead contain the average speed with which the application should execute starting from t_{now} to achieve its maximum achievable relative performance, and the value of the maximum relative performance, respectively.

For a given ω_g, there exist two values k and $k+1$ such that:

$$\sum_m W_{k,m} \leq \omega_g \leq \sum_m W_{k+1,m} \tag{6}$$

Allocating a CPU power of ω_g to all jobs will result in a relative performance u_m for each job m in the range $V_{k,m} \leq u_m \leq V_{k+1,m}$. That corresponds to a hypothetical CPU allocation ω_m in the range $W_{k,m} \leq \omega_m \leq W_{k+1,m}$.

At some point the algorithm needs to know the relative performance that each application will achieve (u_m) if it decides to allocate a CPU power of ω_g to all applications combined. We must find values ω_m and u_m for each application m such that equation 6 is satisfied and that fall within the ranges described above. It must also be satisfied that $\sum_m \omega_m = \omega_g$. As finding a solution for this final requirement implies solving a system

of linear equations, which is too costly to perform in an on-line placement algorithm, we use an approximation based on the interpolation of ω_m from cells $W_{k,m}$ and $W_{k+1,m}$, where k and $k+1$ follow equation 6, and deriving u_m from ω_m. This technique is not included here because of space constraints, but is described in detail in [24].

Evaluating placement decisions Let P be a given placement. Let ω_m be the amount of CPU power allocated to application m in placement P. For applications that are not placed, $\omega_m = 0$.

To calculate the relative performance of application m given a placement P calculated at time t_{now} for a control cycle of length T, we calculate a hypothetical relative performance function at time $t_{\text{now}} + T$. For each job, we increase its α^* by the amount of work that will be done over T with allocation ω_m. We use thus obtained hypothetical relative performance to extrapolate u_m from matrices W and V for $\omega_g = \sum_m \omega_m$.

Thus, we use the knowledge of placement in the next cycle to predict the job's progress over the cycle's duration, and we use the hypothetical function to predict job performance in the following cycles. We also assume that the total allocation to long-running workload in the following cycles will be the same as in the next cycle. This assumption helps us balance batch work execution over time.

4.3 Hypothetical relative performance: an illustrative example

In this example (created using the simulator discussed in more detail in Section 5) we illustrate how the hypothetical relative performance guides our algorithm to make placement decisions.

We use three jobs, J1, J2, and J3, with properties shown in Table 1, and a single node with 2000MB of RAM and one 1000MHz processor. The memory characteristics of the jobs and the node mean that the node can host only two jobs at a time. J1 can completely consume the node's CPU capacity, whereas J2 and J3, at maximum speed, can each consume only half of the node's CPU capacity.

We execute two scenarios, S1 and S2, which differ in the setting of the completion time factor for J2, which in turn affects the completion time goal for J2, as illustrated in Table 1. Note that J3 has a completion time factor of 1, which means that in order to meet its goal it must be started immediately after submission and that it must execute with the maximum speed throughout its life. To improve the clarity of mathematical calculations, we also use unrealistic execution times (in the order of seconds) and a control cycle $T = 1s$.

					Scenario 1			**Scenario 2**		
Job property	**J1**	**J2**	**J3**	**Job property**	**J1**	**J2**	**J3**	**J1**	**J2**	**J3**
Start time [s]	0	1	2	Relative goal factor	5	**4**	1	5	**3**	1
Max speed [MHz]	1,000	500	500	Relative goal [s]	20	**16**	8	20	**12**	8
Mem requirement [MB]	750	750	750	Completion time goal [s]	20	**17**	10	20	**13**	10
Work [Mcycles]	4,000	2,000	4,000							
Min execution time [s]	4	4	8							

Table 1. Hypothetical Relative Performance Example: System Properties

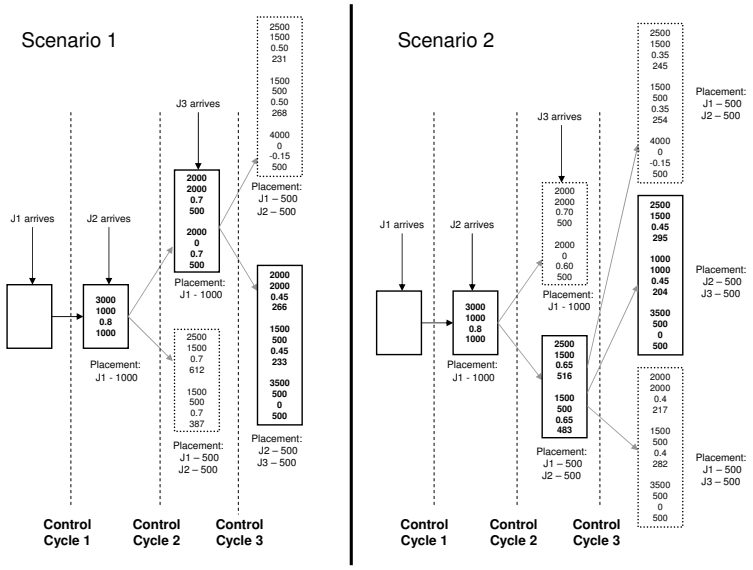


Fig. 1. Hypothetical Relative Performance Example: Cycle-by-cycle execution. Rectangular boxes show the outstanding work, $\alpha_m - \alpha_m^*$, work done, α_m^*, value of hypothetical relative performance and corresponding CPU allocation for each job currently in the system.

Figure 1 shows cycle-by-cycle executions of the algorithm for S1 and S2, respectively, illustrating relevant placement alternatives that are considered in successive control cycles. The boxes with solid outlines show the choices that the algorithm makes, and those with dotted outlines indicate viable alternatives that are not chosen. The reasoning for the choices made is described below.

In cycle 1 of S1 and S2, only one job, J1, is in the system, hence the only reasonable placement is the one that allocates the maximum speed to J1. After the arrival of J2, in cycle 2, two placements are considered: P1, in which both J1 and J2 are running with an allocated speed of 500 MHz (for J1 this amounts to 50% reduction of the capacity allocated in cycle 1), and P2, in which J2 is not placed and J2 continues to run at maximum speed. In S1, P1 and P2 have the same relative performance value of 0.7 to both applications. Therefore, P2 is selected, since it does not require any placement changes. In S2, due to the tightened goal of J2, the utilities of P1 and P2 are (0.65, 0.65) and (0.6, 0.7), respectively, where (x, y) is an increasingly ordered vector of utilities for J1 and J2. Therefore, in S2, P1 is better choice as it equalizes the relative distance of the performance of J1 and J2 from their respective goals.

The difference in the value of hypothetical utilities between S1 and S2, can be illustrated using J2 as an example. If J2 is not started in cycle 2, and hence is started in cycle 3 or later, its earliest possible completion time is 19. In S1, this results in a maximum achievable relative performance of 0.65 ($\approx (16-5)/16$), whereas in S2, it is only 0.6 ($\approx (12-5)/12$).

Since in cycle 2 of S1 and S2 the algorithm has made different decisions, from this point the scenarios diverge. However, a similar rationale may be used to explain the decisions made by the algorithm in cycle 3, also shown in Figure 1.

5 Experiments

In this section we present three experiments performed using a simulator previously used and validated in [25] and [18].

The simulator implements a variety of scheduling policies for batch jobs and also includes a performance model for transactional workloads, as described in Section 3.3. It assumes a virtualized system, in which VM control mechanisms such as suspend, resume, and live migration are used to configure application placement. The costs of these mechanisms, in terms of the time they take to complete, are obtained from measurements taken using a popular virtualization product for Intel-based machines. These measurements reveal simple linear relationships between the VM memory footprint and the cost of the operation, and can be described as Suspend Cost $=$ VM Footprint $* 0.0353s$, Resume Cost $=$ VM Footprint $* 0.0333s$, Migrate Cost $=$ VM Footprint $* 0.0132s$. The boot time observed for all our virtual machines was $3.6s$.

For the purpose of easily controlling the tightness of SLA goals, we introduce a relative goal factor which is defined as the ratio of the relative goal of the job to its execution time at the maximum speed, $\frac{\tau_m - \tau_m^{\text{start}}}{t_m^{\text{best}}}$.

In the experiments, we first study the effectiveness of our technique in handling a homogeneous workload. This paper focuses on batch workload, as the benefits of our approach in managing transactional workload have been shown previously [18]. This permits us to study the algorithm's behavior with a reduced number of variables, while also providing an opportunity to compare our techniques to existing approaches. In the final experiment, we evaluate the effectiveness of our technique in managing a heterogeneous mix of transactional and long-running workloads.

5.1 Experiment One: Relative performance prediction accuracy

In this experiment, we examine the basic correctness of our algorithm by stressing it with a sequence of identical jobs—jobs with the same profiles and SLA goals. We use this scenario to examine the accuracy with which hypothetical relative performance predicts the actual job performance.

When jobs are identical, the best scheduling strategy is to make no placement changes (suspend, resume, migrate). This is because there is no benefit to job completion times (when considered as a vector) to be gained by interrupting the execution of a currently placed job in order to place another job.

We consider a system of 25 nodes, each of which has four 3.9GHz processors and 16GB of RAM. We submit to this system 800 identical jobs with properties shown in Table 2. Jobs are submitted to the system using an exponential inter-arrival time distribution with an average inter-arrival time of 260s. This arrival rate is sufficient to cause queuing at some points during the experiment. The control cycle length is 600s.

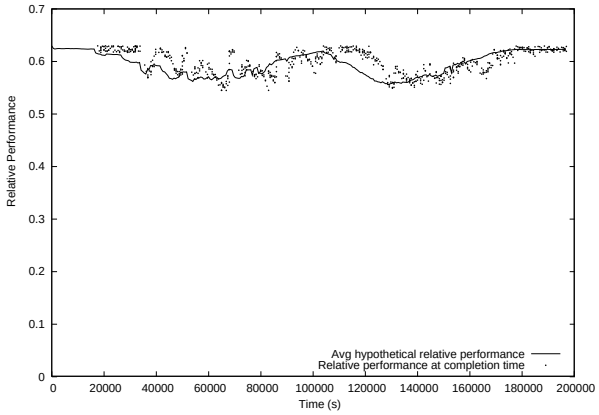


Fig. 2. Experiment One: Average hypothetical relative performance over time and actual relative performance achieved at completion time

Observe that each job's maximum speed permits it to use a single processor, and so four jobs could run at full speed on a single node. However, the memory characteristics restrict the number of jobs per node to three. Consequently, no more than 75 jobs can run concurrently in the system. Each job, running at maximum speed, takes 17,600s to complete. The relative goal factor for each job is 2.7, resulting in a completion time goal of 47,520s ($2.7 * 17,600$), as measured from the submission time.

The maximum achievable relative performance for a job described in Table 2 is 0.63. This relative performance will be achieved for a job that is started immediately upon submission and runs at full speed for 17,600s. In that case, the job will complete 29,920s before its completion time goal, and thus will have taken 37% of the time between the submission time and the completion time goal to complete. This relative performance is an upper bound for the job, and will be decreased if queuing occurs.

In Figure 2, we show the average hypothetical relative performance over time as well as the actual relative performance achieved by jobs at completion time. When no jobs are queued, the hypothetical relative performance is 0.63 and it decreases as more jobs are delayed in the queue. Notice that the relative performance achieved by jobs at their completion time has a shape similar to that of the hypothetical relative performance, but is shifted in time by about 18000 sec. This is expected, as the hypothetical relative performance is predicting the actual relative performance that jobs will obtain at the time they complete, as thus is affected by job submissions, while the actual relative performance is only observed at job completion. The algorithm does not elect to suspend or migrate any jobs during this experiment, hence we do not include a figure showing the number of placement changes performed. Finally, we evaluated the execution time for the algorithm at each control cycle when running on a 3.2GHz Intel Xeon node. In normal conditions, the algorithm produces a placement for this system in about 1.5s. We also observed that when all submitted jobs can be placed concurrently, the algorithm is able to take internal shortcuts, resulting in a significant reduction in execution time.

Property	Value	Property	Value
Maximum speed [MHz]	3,900 (1 CPU)	Minimum execution time [s]	17,600
Memory requirement [MB]	4,320	Relative goal factor	2.7
Work [Mcycles]	68,640,000	Relative goal [s]	**47,520**

Table 2. Properties of Experiment One

5.2 Experiment Two: Comparing different scheduling algorithms

In this experiment, we compare our algorithm with alternative scheduling algorithms. We do so in a system presented with jobs with varying profiles and SLA goals. The relative goal factors for jobs are randomly varied among values 1.3, 2.5, and 4 with probabilities 10%, 30%, and 60%, respectively. The job minimum execution times and maximum speeds are also randomly chosen from three possibilities—9,000s at 3,900MHz, 17,600s at 1,560MHz, and 600s at 2,340MHz which are selected with probabilities 10%, 40%, and 50%, respectively.

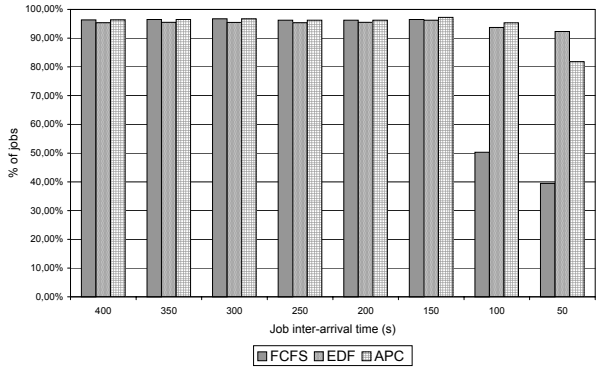


Fig. 3. Experiment Two: Percentage of jobs that met the deadline

We compare our algorithm (referred to as APC) with simple, effective, and well-known scheduling algorithms: Earliest Deadline First (EDF) and First-Come, First-Served (FCFS). Note that while EDF is a preemptive scheduling algorithm, FCFS does not preempt jobs. In both cases, a first-fit strategy was followed to place the jobs.

We use eight different inter-arrival times, ranging from 50s to 400s, and continue to submit jobs until 800 have completed. The experiment is repeated for the three algorithms: APC, EDF, and FCFS. In the paper we concentrate on the results for inter-arrival times of 200s and 50s due to space limitations (see [24] for more results).

Figure 3 shows the percentage of jobs that met their completion time goal. There is no significant difference between the algorithms when inter-arrival times are greater than 100s—this is expected, as the system is underloaded in this configuration. However, with an inter-arrival period of 100s or less, FCFS starts struggling to make even 50% of the jobs meet their goals. EDF and APC have a significantly higher, and comparable, ratio of jobs that met their goals. At a 50s inter-arrival time, the goal satisfaction

rate for FCFS has dropped to 40%, and the goal satisfaction rate is actually higher for EDF than for APC. However, Figures 4 and 5 show the penalties for EDF's slightly (10%) higher satisfaction rate.

Figure 4 shows that one of these penalities is that EDF makes considerably more placement changes than does the APC once the inter-arrival time is 150s or less. Recall that FCFS is non-preemptive, and so makes no changes. Note that in this experiment, we did not consider the cost of the various types of placement changes—this does not change the conclusions, as our technique is making many fewer changes that EDF under heavy load. This figure, coupled with Figure 3, shows our algorithm's ability to making few changes to the system whilst still achieving a high on-time rate.

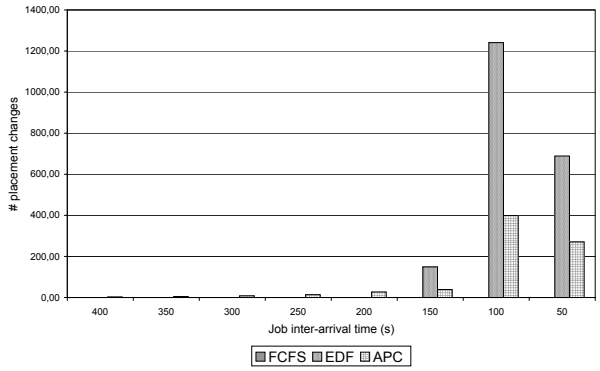
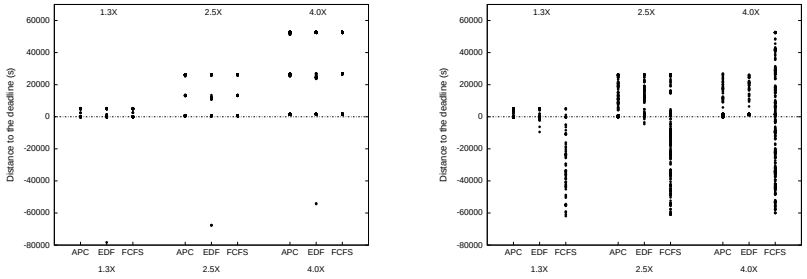


Fig. 4. Experiment Two: Number of jobs migrated, suspended, and moved_and_resumed



(a) 200s (b) 50s

Fig. 5. Experiment Two: distribution of distance to the goal at job completion time, for two different mean inter-arrival times (50s and 200s)

Figure 5 shows the distribution of distance to the deadline at job completion time for the three different relative goal factors (1.3, 2.5 and 4.0). We show these results for inter-arrival times of 200 and 50 seconds, in Figure 5 (a) and (b), respectively. Points with distance to the goal greater than zero indicate jobs that completed before their goal. Ob-

serve that for inter-arrival times of 200s, all three algorithms are capable of making the majority of jobs meet their goal, and the points for each algorithm are concentrated—for each algorithm and each relative goal factor, the distance points form three clusters, one for each minimum execution time. However, at an inter-arrival time of 50s, the algorithms produce different distributions of distances to the goal. In particular, observe that for APC the data points are closer together than for EDF (this is most easily observed for the relative goal factor of 1.3). This illustrates that APC outperforms EDF in equalizing the satisfaction of all jobs in the system.

5.3 Experiment Three: Heterogeneity

In this experiment, we examine the behavior of our algorithm in a system presented with heterogeneous workloads. We demonstrate how our integrated management technique is applicable to combined management of transactional and long-running workloads. The experiment will show how our algorithm allocates resources to both workloads in a way that equalizes their satisfaction in terms of distance between their performance and performance goals. We compare our dynamic resource sharing technique to a static approach in which resources are not shared, and are pre-allocated to one type of work. This static approach is widely used today to run mixed workloads in datacenters.

We extend Experiment One by adding transactional workload to the system, and compare three different system configurations subject to the same mixed workload. In the first configuration we use our technique to perform dynamic application placement with resource sharing between transactional and long-running workloads. In the second and third configurations we consider a system that has been partitioned into two groups of machines, each group dedicated to either the transactional or the long-running workload. In both configurations, we use a First-Come First-Served (FCFS) to place jobs—FCFS was chosen because it is a widely adopted policy in commercial job schedulers. Notice that creating static system partitions is a common practice in many datacenters. In the second configuration, we dedicate 9 nodes to the transactional workload (9 nodes offer enough CPU power to fully satisfy this workload), and 16 nodes to the long-running workload. In the third configuration, we dedicate 6 nodes to the transactional application and 19 to the long-running workload.

To simplify the experiment, the transactional workload is handled by a single application, and is kept constant throughout. Note that the long-running workload is exactly the same as that presented in Section 5.1. The memory demand of a single instance of the transactional application was set to a sufficiently low value that one instance could be placed on each node alongside the three long-running instances that fit on each node in Experiment One. This was done to ensure that the two different types of workload compete only for CPU resources (notice from Experiment One that a maximum of 3 long-running instances can be placed in the same node because of memory constraints).

The relative performance of transactional workloads is calculated as described in Section 3.3. A relative performance of zero means that the actual response time exactly meets the response time goal: lower relative performance values indicate that the response time is greater than the goal (the requests are being serviced too slowly), and higher relative performance values indicate that the response time is less than the goal (the requests are being serviced quickly). In this experiment, the maximum achievable

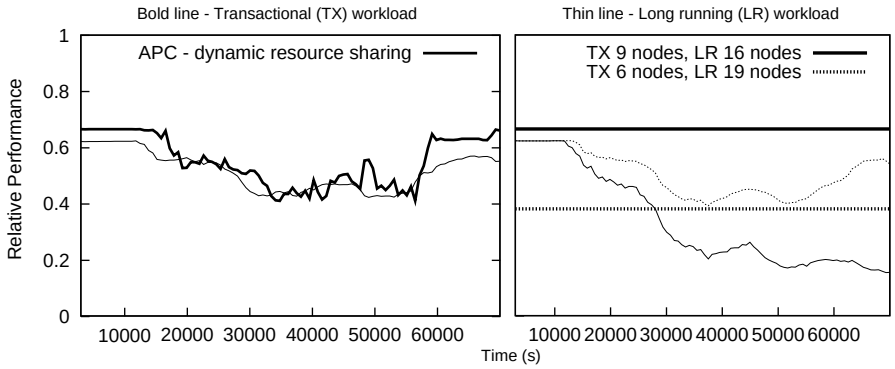


Fig. 6. Experiment Three: actual relative performance for the transactional workload and average calculated hypothetical relative performance for the long-running workload

relative performance for the transactional workload is around 0.66, at an approximate allocation of 130,000MHz. Allocating CPU power in excess of 130,000MHz to this application will not further increase its satisfaction: that is, it will not decrease the response time. This is normal behavior for transactional applications—the response time cannot be reduced to zero by continually increasing the CPU power assigned. Notice that 130,000MHz is less than to the CPU capacity of 9 nodes, and so the transactional workload can be completely satisfied by 9 nodes.

The experiment starts with a system subject to the constant transactional workload used throughout, in addition to a small (insignificant) number of long-running jobs already placed. Then, we start submitting a number of long-running jobs using an inter-arrival time short enough to produce some job queueing. As more long-running jobs are submitted, following the workload properties described in Section 5.1, more CPU demand is observed for the long-running workload. In the end of the experiment, the long-running job inter-arrival time is increased to a value high enough to expect that the job queue length will start decreasing.

Figures 6 and 7 show the results obtained for the three system configurations described above. Figure 6 shows the relative performance achieved by both the transactional and long-running workload for each of the configurations. For the transactional workload we show actual relative performance, and for the long-running workload we show hypothetical relative performance, described in Section 4.2. Although hypothetical relative performance is a predicted value, previous experiments have already shown that this approximation is accurate enough for performance prediction purposes. In addition, we verified for this particular experiment that the utilities achieved by jobs at completion time, long after they were submitted and placements calculated, met their predicted performance. Figure 7 shows the CPU power allocated to each workload over the experiment.

Looking at the results for our dynamic resource sharing technique it can be observed that at the beginning of the experiment the transactional application gets as much CPU power as it can consume, as there is little or no contention with long-running jobs—obtaining its maximum achievable relative performance of 0.66. As more long-running jobs are submitted, the hypothetical relative performance for those long-running jobs starts to decrease as the system becomes increasingly crowded. As soon as the hypothetical relative performance calculated for the long-running jobs becomes lower that the

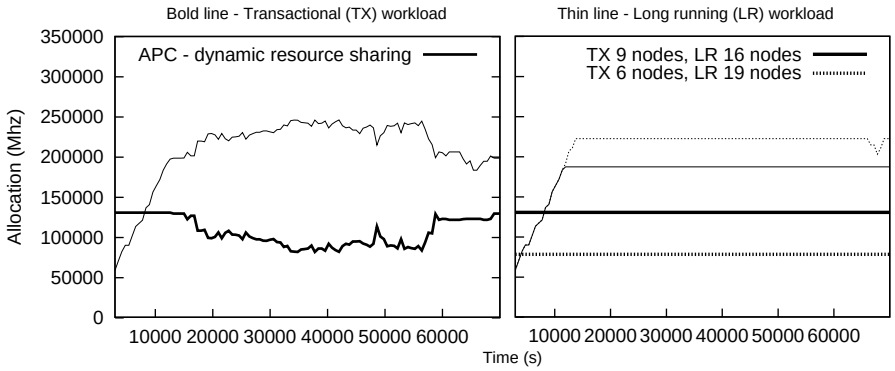


Fig. 7. Experiment Three: CPU power allocated to each workload for the three system configurations

relative performance observed for the transactional workload (that is to say, no more resources can be allocated to the long-running workload without taking CPU power away from the transactional workload), our algorithm starts to reduce the allocation for the transactional workload and give that CPU power instead to the long-running workload, until the relative performance each achieves is equalized. At the end of the experiment, the job submission rate is slightly decreased, which results in more CPU power being returned to the transactional workload again. The relative performance observed for both workloads is continuously adjusted by dynamically allocating resources over time. The result is that the resource sharing between both workloads is dynamically and unevenly adjusted, but achieving a similar relative performance for each workload, what is the main purpose of our proposed technique.

The results for the static system configurations reveal that the overall performance they deliver is lower than the performance observed for our dynamic resource sharing technique, and the performance of both static and dynamic approaches is only comparable when the size of each machines partition exactly matches the resource allocation decided by our technique. Notice that when 9 nodes are dedicated to the transactional workload (offering more than the CPU power required to fully satisfy it), the relative performance achieved by the transactional workload is, as expected, 0.66—the maximum achievable. In this configuration, while transactional workload obtains good performance, long-running jobs struggle to meet their completion time goals, as shown by the low achieved relative performance values. When only 6 nodes are dedicated to the transactional workload, the relative performance that it achieves is consistently lower than that achieved with our dynamic resource sharing technique, while the performance benefits observed for the long-running jobs are not obvious when compared to the results obtained with our technique. Recall also that relative performance represents the relative distance to the goal achieved by each particular workload—distance to the response time goal for the transactional application and distance to the completion time goal for long-running jobs. Thus, relative performance is a direct measurement of the performance obtained by each workload.

6 Conclusions and future work

In this paper we present a technique that allows integrated management of heterogeneous workloads composed of transactional applications and long-running jobs, dynamically placing the workloads in such a way as to equalize their satisfaction. We use relative performance functions to make the satisfaction and performance of both workloads comparable. We formally describe the technique, and then demonstrate that it not only performs well in presence of heterogeneous workloads but it also shows consistent performance in presence only of long-running jobs compared to other well-known scheduling algorithms. We perform our experiments with a simulator already used and validated against a system prototype in [25, 18]. While here we mainly focus on the description and evaluation of the management of long-running jobs, transactional workloads were extensively covered in [18]. We expect to extend this technique in the future to offer explicit support for parallel jobs, and we also need to work on the on-the-fly generation of job profiles. The optimization technique could also be extended to focus on resources other than CPU.

Acknowledegments

This work is partially supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625 and by the BSC-IBM collaboration agreement SoW Adaptive Systems.

References

1. Sun Microsystems: Behavior of mixed workloads consolidated using Solaris Resource Manager software. Technical report (May 2005)
2. Lin, B., Dinda, P.: VSched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In: Proc. ACM/IEEE Supercomputing, Seattle, WA (Nov. 2005)
3. Jensen, E.D., Locke, C.D., Tokuda, H.: A time-driven scheduling model for real-time operating systems. In: IEEE Real-Time Systems Symposium. (1985) 112–122
4. Ravindran, B., Jensen, E.D., Li, P.: On recent advances in time/utility function real-time scheduling and resource management. In: ISORC '05: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05), Washington, DC, USA, IEEE Computer Society (2005) 55–60
5. Balli, U., Anderson, J.S.: Utility accrual real-time scheduling under variable cost functions. IEEE Trans. Comput. **56**(3) (2007) 385–401 Member-Haisang Wu and Senior Member-Binoy Ravindran and Member-E. Douglas Jensen.
6. David Vengerov, Lykomidis Mastroleon, D.M., Bambos, N.: Adaptive data-aware utility-based scheduling in resource-constrained systems. Sun Technical Report TR-2007-164, Sun Microsystems (April 2007)
7. Lee, C.B., Snavely, A.E.: Precise and realistic utility functions for user-centric performance analysis of schedulers. In: HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing, New York, NY, USA, ACM (2007) 107–116
8. Chun, B.N., Culler, D.E.: User-centric performance analysis of market-based cluster batch schedulers. In: CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, Washington, DC, USA, IEEE Comp. Society (2002) 30

9. Foster, I., Freeman, T., Keahey, K., Scheftner, D., Sotomayor, B., , Zhang, X.: Virtual clusters for grid communities, Singapore (May 2006)
10. Czajkowski, G., Wegiel, M., Daynes, L., Palacz, K., Jordan, M., Skinner, G., Bryce, C.: Resource management for clusters of virtual machines, Cardiff, UK (May 2005) 382–389
11. Zhu, X., Wang, Z., Singhal, S.: Utility-driven workload management using nested control design. American Control Conference, 2006 (14-16 June 2006) 6 pp.–
12. Padala, P., Shin, K.G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A., Salem, K.: Adaptive control of virtualized resources in utility computing environments. In: EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, New York, NY, USA, ACM (2007) 289–302
13. Wang, Z., Zhu, X., Padala, P., Singhal, S.: Capacity and performance overhead in dynamic resource allocation to virtual containers. Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on (May 21 2007-Yearly 25 2007) 149–158
14. Kellerer, H., Pferschy, U., Pisinger, D.: Knapsack Problems. (2004)
15. Karve, A., Kimbrel, T., Pacifici, G., Spreitzer, M., Steinder, M., Sviridenko, M., Tantawi, A.: Dynamic placement for clustered web applications. In: WWW Conference, Edinburgh, Scotland (May 2006)
16. Kimbrel, T., Steinder, M., Sviridenko, M., Tantawi, A.: Dynamic application placement under service and memory constraints. In: International Workshop on Efficient and Experimental Algorithms, Santorini Island, Greece (May 2005)
17. Wang, X., Lan, D., Wang, G., Fang, X., Ye, M., Chen, Y., Wang, Q.: Appliance-based autonomic provisioning framework for virtualized outsourcing data center. In: ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing, Washington, DC, USA, IEEE Computer Society (2007) 29
18. Carrera, D., Steinder, M., Whalley, I., Torres, J., Ayguadé, E.: Utility-based placement of dynamic web applications with fairness goals. In: 11th IEEE/IFIP Network Operations and Management Symposium (NOMS 2008), Salvador Bahia, Brazil (2008)
19. Xu, J., Zhao, M., Fortes, J., Carpenter, R., Yousif, M.: On the use of fuzzy modeling in virtualized data center management. Autonomic Computing, 2007. ICAC '07. Fourth International Conference on (11-15 June 2007) 25–25
20. Urgaonkar, B., Shenoy, P., Roscoe, T.: Resource overbooking and application profiling in shared hosting platforms. In: Proc. Fifth Symposium on Operating Systems Design and Implementation, Boston, MA (Dec. 2002)
21. Pacifici, G., Spreitzer, M., Tantawi, A., Youssef, A.: Performance management for cluster-based web services. IEEE Journal on Selected Areas in Communications **23**(12) (Dec. 2005)
22. Pacifici, G., Segmuller, W., Spreitzer, M., Steinder, M., Tantawi, A., Youssef, A.: Managing the response time for multi-tiered web applications. Technical Report Tech. Rep. RC 23651, IBM (2005)
23. Pacifici, G., Segmuller, W., Spreitzer, M., Tantawi, A.: Dynamic estimation of cpu demand of web traffic. In: VALUETOOLS, Pisa, Italy (Oct. 2006)
24. Carrera, D., Steinder, M., Whalley, I., Torres, J., Ayguadé, E.: Managing SLAs of heterogeneous workloads using dynamic application placement. Technical Report RC 24469, IBM Research (Jan. 2008)
25. Steinder, M., Whalley, I., Carrera, D., Gaweda, I., Chess, D.: Server virtualization in autonomic management of heterogeneous workloads. In: 10th IEEE/IFIP Symposium on Integrated Management (IM 2007), Munich, Germany (2007)