RC24469 (W0801-081) January 22, 2008
Computer Science

IBM Research Report

Managing SLAs of Heterogeneous Workloads Using Dynamic Application Placement

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

[1]Technical University of Catalonia (UPC)
Barcelona Supercomputing Center (BSC)
Barcelona, Spain

[2]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Managing SLAs of heterogeneous workloads using dynamic application placement

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

In this paper we address the problem of managing heterogeneous workloads in a virtualized data center. We consider two different workloads: transactional applications and long-running jobs. We present a technique that permits collocation of these workload types on the same physical hardware. Our technique dynamically modifies workload placement by leveraging control mechanisms such as suspension and migration, and strives to optimally trade off resource allocation among these workloads in spite of their differing characteristics and performance objectives. Our approach builds upon our previous work on dynamically placing transactional workloads. This paper extends our framework with the capability to manage long-running workloads. We achieve this goal by using utility functions, which permit us to compare the performance of various workloads, and which are used to drive allocation decisions. We demonstrate that our technique maximizes heterogeneous workload performance while providing service differentiation based on high-level performance goals.

1 Introduction

Many organizations rely on a heterogeneous set of applications 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, today they are typically run on separate dedicated hardware and managed using workload specific management software. Such separation adds to the complexity of data center management and reduces the flexibility of resource allocation. Therefore, organizations demand management solutions that permit these kinds of workloads to run on the same physical hardware and be managed using the same management software.

The management of heterogeneous workloads introduces a number of challenges in the area of their 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. Virtualization technologies also enable separation between management concerns, permitting software and configuration tasks inside virtual machines to be done a priori. The runtime management system is only responsible for the runtime performance and availability of virtualized workloads.

Integrated performance management of heterogeneous 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 certain time interval, while performance goals for non-interactive workloads concern the performance (e.g., completion time) of individual jobs. Second, due to the nature of their performance goals and short duration of individual requests, interactive workloads lend themselves to automation at short control cycles. Non-interactive workloads typically require calculation of a schedule for an extended period of time.

This paper proposes a solution to allocate resources among transactional web workloads and long-running compute intensive jobs. The objective of the approach is to provide fair differentiation of performance among all workloads in response to varying workload intensities.

To our knowledge, our work is the first proposal that combines an explicit support for heterogeneous workloads in virtualized environments, using a utility-driven scheduling mechanism with fairness goals. A preliminary working prototype of our proposal that made use of a commercial middleware to enforce its decisions was described in [20]. The performance management aspects pertaining to transactional workloads were introduced in [2]. The original contribution of this paper is a scheme for modeling the performance of, and managing, 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 manage heterogeneous workloads using utility-driven resource allocation. In Section 4, we introduce the process of calculating the utility function for long-running applications. In Section 5 we evaluate our approach via simulation.

2 Related work

In this paper we present a technique that allows the management to high-level goals of collocated long-running and transactional workloads in virtualized environments. We use utility functions to model the satisfaction of both long-running jobs and transactional workloads for a particular resource allocation – the different types of workload have different characteristics, and different performance goals, and utility functions offer a mechanism to make their performance comparable. We run both workloads inside virtual machines, in order to properly manage their performance, and our management also exploits the clustering nature of transactional workloads.

Both the use of utility functions for workload management, and managing clusters of virtual machines, are areas already studied in the literature, but our approach is the first one that combines them to successfully manage heterogeneous workloads with fairness goals. This paper will not discuss the management of transactional workloads further – the reader is referred to our earlier work, in particular [20, 2].

The explicit management of heterogeneous workloads was previously studied in [21], in which a number of CPU shares were manually allocated to run mixed workloads on a large multiprocessor system. This was a static approach, and did not run workloads within virtual machines. Virtuoso [14] 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 is broader as it addresses resource allocation for heterogeneous workloads across a cluster of physical machines.

The use of utility-driven strategies to manage workloads was first introduced in the scope of 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 [8], 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, including memory- [6] and energy-aware [25] utility-driven scheduling, are summarized in [19] with special focus on real-time embedded systems. In [1], the authors present a utility-driven scheduling mechanism that aims to maximize the aggregated system utility. In contrast, our technique does not focus on real-time systems, but on any general system for which performance goals can be expressed as utility functions. In addition, we introduce the notion of fairness into our application-centric management technique – our objective is not to maximize the system utility, but instead to at least maximize the lowest utility across long-running jobs and transactional applications present in the system.

Outside of the realm of the real-time systems, the authors of [5] 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 utility functions.

In our work we use monotonic and continuous utility functions to represent the satisfaction of both transactional and long-running workloads, but other approaches have been studied in the literature. In [13], the authors discuss the best shape for the utility functions (extending the work presented in [12]). The authors of [3] use user-defined utility functions to represent the value of resources, and their market-based batch scheduler is driven by these utility functions to allocate resources.

There is also some previous work in the area of managing workloads in virtual machines. Management of clusters of virtual machines is addressed in [7] and [4]. The authors of [7] address the problem of deploying a cluster of virtual machines with given resource configurations across a set of physical machines. Czajkowski et al. [4] define an API for a Java VM that permits a developer to define resource allocation policies. In [27] and [18], a two-level control loop is proposed to make resource allocation decisions within a single physical machine, but does not address integrated management of a collection of physical machines. The authors of [24] study the overhead of a dynamic allocation scheme that relies on virtualization as opposed to static resource allocation. Their evaluation covers CPU-intensive jobs as well as 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 presence of resource contention.

Placement problems in general (either in the presence of virtualization technologies or not) have also been studied in the optimization literature, frequently using techniques including bin packing, multiple knapsack problems, and multi-dimensional knapsack problems [10]. The optimization problem that we consider presents a non-linear optimization objective while previous approaches [9, 11] to similar problems address only linear optimization objectives. In [23], the authors evaluate a similar problem to that addressed in our work (but restricted to transactional applications), and use a simulated annealing optimization algorithm. Their optimization strategy aims to maximize the overall system utility while we focus on first maximizing the lowest utility across applications, which increases fairness and prevents starvation, as was shown in [2]. In [26], a fuzzy logic controller is implemented to make dynamic resource management decisions. This approach is not application-centric – it focuses instead on global throughput – and considers only transactional applications. The algorithm proposed in [22] 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 from the queue 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 [17, 15] by queuing requests that cannot be immediately accommodated by server nodes. A separate component, called the *work profiler* [16], 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 *application placement controller* (APC).

Long-running 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 the APC as an advisor as to where and when a job should be executed. When the APC makes a decision, actions pertaining to long-running 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 a utility function 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, the APC examines the placement of applications on nodes and their resource allocations, evaluates the utility 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, Γ_n and Ω_n. With each application, we associate its load independent demand, γ_m that represents the amount of memory consumed by this application whenever it is started on a node. The CPU requirements of applications are given in the form of utility functions.

A utility function for a given application expresses the satisfaction of that application from a given resource allocation. For comparability among workloads, utility functions of all applications must obey a common contract. In our system, we assume that utility functions are monotonically increasing and range from $-\infty$ to 1, and have 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.

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 utility function for each application may be expressed as a function of L.

The objective of APC, in each control cycle, is to find the best possible new placement of applications. 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 utility application cannot be allocated any more resources), the system should continue improving the utility of other applications.

The APC finds a placement that meets the above objective while ensuring that neither memory nor CPU capacity of any node is overloaded. In addition, APC employs heuristics that aim to minimize the number of placement changes compared to the placement currently in effect. While finding the optimal placement, APC also observes a number of constraints, such as resource constraints, collocation constraints and application pinning, amongst others.

3.2.1 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 [2].

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 while 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 utility. In the process, the algorithm examines application utility asking the following questions:

- What is the utility 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 utility value?

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

3.3 Transactional workloads

In our system, a user can associate a response time goal, τ_m with each transactional application. (In fact, we can associate such goals with a smaller granularity management units, flows, but omit this detail from this paper.) 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 [17]:

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

We leverage the performance model of the request router as well as 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 utility u by taking the inverse function of u_m, $\omega_m(u)$.

4 Long-running 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 long-running jobs as individual management entities, as their completion times are not independent. For example, if jobs that are currently running complete faster, this permits jobs currently in the queue (not running) to complete faster as well. Thus, performance predictions for long-running jobs must be done in relation other 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 us to calculate 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 avoid this complexity by proposing 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 is 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, $\omega_{k,m}^{\max}$. A CPU allocation higher by $\omega_{k,m}^{\max}$ would not be consumed in this stage.
- The minimum speed with which the stage must run, whenever it runs, $\omega_{k,m}^{\min}$. An allocation lower than $\omega_{k,m}^{\min}$ would not permit a correct execution of the stage.
- The memory requirement $\gamma_{k,m}$

Performance objectives. An 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 a utility function that maps actual job completion time t_m to a measure of satisfaction from achieving it, $u_m(t_m)$. Many utility function forms may be used. In our implementation, we use 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.
- CPU time consumed thus far, α_m^*

4.2 Stage aggregation in a control cycle

We now focus on the reasoning that the APC must apply in order to decide which jobs should be scheduled for execution, on which nodes they should execute, and how much CPU power they should be allocated. We presume that the APC operates with a control cycle of duration T. Thus, when making decisions at time t_{now}, the APC must consider job progress between time t_{now} and time $t_{\text{now}} + T$.

Depending on the stage duration and the value of T, one or more stages can be executed in a control cycle. Since resource allocation will not change for the duration of the control cycle, the resource allocation must be such so as to accommodate all stages that will execute in this cycle.

Considering this, in addition to the job characteristics introduced in Section 4.1, we must now define some additional parameters.

First, we define the cumulative work that a job must complete, $\alpha_{D,m}^c = \sum_{i=1}^{D,m} \alpha_{i,m}$. Since α_m^* cycles have already been completed, the remaining work to complete D stages is $\alpha_{D,m}^{cr} = \max(0, \alpha_{D,m}^c - \alpha_m^*)$. The remaining work to complete the entire job is simply $\alpha_{N_m,m}^{cr}$. At t_{now}, the job must have already completed D_m^{done} stages which may be obtained by taking $D_m^{\text{done}} = \max_D \alpha_{D,m}^c \leq \alpha_m^*$. For each job stage, we can now obtain the work remaining in this stage, which is given as follows:

$$\alpha_{D,m}^r = \begin{cases} 0 & \text{if } D \leq D_m^{\text{done}} \\ \alpha_{D,m}^c - \alpha_m^* & \text{if } D = D_m^{\text{done}} + 1 \\ \alpha_{D,m} & \text{otherwise} \end{cases} \tag{3}$$

Let us assume that in each state a job is allocated the maximum usable speed. Then, the time remaining to complete stage D is $t_{D,m}^r = \frac{\alpha_{D,m}^r}{\omega_{D,m}^{max}}$. In time T, the job cannot progress to complete more than D_m^{last} stages where D_m^{last} is the maximum D such that $\sum_{i=D_m^{\text{done}}+1}^{D} t_{i,m}^r \leq T$.

We can now make a conservative assumption, that throughout the cycle that starts at t_{now} and lasts for time T, the job will require the minimum CPU speed, maximum CPU speed, and memory of $\omega_m^{\min}$, ω_m^{req}, and γ_m, respectively, which are defined as follows.

$$\omega_m^{\min} = \max_{D_m^{\text{done}}+1 \leq i \leq D_m^{\text{last}}} \omega_i^{\min} \tag{4}$$

$$\omega_m^{\text{req}} = \max_{D_m^{\text{done}}+1 \leq i \leq D_m^{\text{last}}} \omega_i^{\max} \tag{5}$$

$$\gamma_m = \max_{D_m^{\text{done}}+1 \leq i \leq D_m^{\text{last}}} \gamma_i \tag{6}$$

4.3 Maximum achievable utility

There is an inherent upper bound to the utility that a job may achieve considering its progress thus far. When progressing with the maximum possible speed, a job will complete in time $t_m^{\text{best}} = \sum_{i=D_m^{done}}^{D} t_{i,m}^r$ thus its completion time will be $t_{\text{now}} + t_m^{\text{best}}$. The maximum utility that the job can achieve given its progress thus far is $u_m^{max} = u_m(t_{\text{now}} + t_m^{\text{best}})$.

At any time, a job may be either running with the maximum speed, running with less than the maximum speed, or stopped or suspended (not running).

In general, in each control cycle in which the CPU allocation of a job is less than the maximum it can use, the value of maximum achievable utility decreases linearly. We illustrate this property with the following example.

Consider a single stage job and suppose that its amount of work and an SLA goal is such that the job can achieve the utility of 0.8 when executing with speed 16,000 MHz. In Figure 1, we show the evolution of the maximum achievable utility over 10 control cycles. We presume that the system cannot allocate more than 16,000 MHz to the job and vary the job maximum speed, ω_m^{req} (plotted on the X axis). When $\omega_m^{req} < 16000$, u_m^{max} is lower than 0.8, and it decreases as ω_m^{req} decreases, but it stays constant over time (plotted on the Y axis), as the system is always able to allocate ω_m^{req}. When $\omega_m^{req} > 16000$, then u_m^{max} may be higher than 0.8, as the job may complete its work faster. However, since the system is not able to provide this much CPU power, and the job only executes with the speed of 16,000 MHz, u_m^{max} decreases over time.

The maximum achievable utility is used to order long-running jobs in the queue. When allocating resources to long-running workload, the APC will first consider jobs with a lower maximum achievable utility.

4.4 Hypothetical utility

To calculate job placement, we need to define a utility function which the APC can use to evaluate its placement decisions. To help answer questions that APC is asking of the utility function for each application we introduce the concept of *hypothetical utility*.

4.4.1 Estimating application utility 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 utility function 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 utility u, $t_m(u) = \tau_m - u(\tau_m - \tau_m^{\text{start}})$. From this number, we can calculate the average speed with which the job must proceed over its remaining lifetime to achieve u, as follows:

$$\omega_m(u) = \frac{\alpha_{N_m,m}^{cr}}{t_m(u) - t_{\text{now}}} \tag{7}$$

To achieve the utility of u for all jobs, the aggregate allocation to all jobs must be $\omega_g = \sum_m \omega_m(u)$. To create the utility function, 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 utilities from now on). We define matrices W and V as follows:

$$W_{i,m} = \begin{cases} \omega_m(u_i) & \text{if } u_i < u_m^{\max} \\ \omega_m(u_m^{\max}) & \text{otherwise} \end{cases} \tag{8}$$

$$V_{i,m} = \begin{cases} u_i & \text{if } u_i < u_m^{\max} \\ u_m^{\max} & \text{otherwise} \end{cases} \tag{9}$$

Cells $W_{i,m}$ and $V_{i,m}$ contain the average speed with which application m should execute starting from t_{now} to achieve utility u_i and value u_i, respectively, if it is possible for application m to achieve utility u_i. If utility 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 utility, and the value of the maximum utility, respectively.

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

$$\sum_m W_{k,m} \le \omega_g \le \sum_m W_{k+1,m} \tag{10}$$

Allocating a CPU power of ω_g to all jobs will result in a utility u_m for each job m in the range:

$$V_{k,m} \le u_m \le V_{k+1,m} \tag{11}$$

That corresponds to a hypothetical CPU allocation in the range:

$$W_{k,m} \le \omega_m \le W_{k+1,m} \tag{12}$$

Figure 2 shows, for two different applications, the allocation required to achieve a range of target utilities, as well as the aggregated demand. Vectors V and W can be constructed by sampling some of the points shown in this figure. Note that, for utilities above the maximum achievable utility for a particular application (points A and B), we take the allocation that corresponds to that maximum achievable utility.

Interpolating u_m and ω_m given ω_g. At some point the algorithm needs to know the utility 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 equations 10, 11, and 12 are satisfied, while also satisfying $\sum_m \omega_m = \omega_g$. 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 10, and deriving u_m from ω_m. Notice from equations 8 and 9 that the value of a cell $V_{i,m}$ does not necessarily correspond to the target utility u_i, and thus a cell $W_{i,m}$ does not necessarily correspond to $\omega_m(u_i)$.

To interpolate ω_m, we first consider the case for which cells $W_{k,m}$ and $W_{k+1,m}$ correspond to the allocations required to make application m achieve utilities u_k and u_{k+1} respectively, i.e., the case for which the calculation of those cells was not constrained by the maximum achievable speed for application m. In this situation, ω_m can be interpolated by calculating first a value $ratio_g$ that corresponds to the position of ω_g relative to the distance between $\sum_m W_{k,m}$ and $\sum_m W_{k+1,m}$. We define $ratio_g$ as:

$$ratio_g = \frac{\omega_g - \sum_m W_{k,m}}{\sum_m W_{k+1,m} - \sum_m W_{k,m}} \tag{13}$$

Once $ratio_g$ is calculated, we can interpolate ω_m as $(W_{k+1,m} - W_{k,m}) * ratio_g + W_{k,m}$.

Figure 3 shows an example of this interpolation. We consider an scenario similar to that shown in Figure 2 in the region indicated by point C. For simplicity, we consider vectors V and W to be filled with values obtained from functions x^3 for job 1, x^2 for job 2 and x^3+x^2 for the aggregated values. We are given a total allocation $\omega_g = 150$ and utility sampling points 2 and 8; and we need to interpolate values $\tilde{\omega}_{job1}$ and $\tilde{\omega}_{job2}$ that satisfy equation 12 and that are as similar as possible. We have the following available data: $W_{2,job1} = 8$, $W_{2,job2} = 4$, and so $\sum_m W_{2,m} = 12$; and $W_{8,job1} = 512$, $W_{8,job2} = 64$, and so $\sum_m W_{8,m} = 576$. We calculate $ratio_g$ to be $(150 - 12)/(576 - 12) = 0.24$. With this value we esttimate $\tilde{\omega}_{job1} = 131$ and $\tilde{\omega}_{job2} = 18$. The corresponding utilities for these job allocations are 4.2 for job 1 and 5.08 for job 2. Obviously, solving the linear system of equations would have produced a more precise solution at a much higher computational cost.

This technique works well when all applications are unconstrained (they have not reached their maximum speed). Notice that the region around point D in Figure 2 shows a different scenario, in which application 1 is running at maximum speed. In this situation, interpolating $\tilde{\omega}_{job1}$ and $\tilde{\omega}_{job2}$ requires some additional effort. Figure 4 shows an scenario similar to that indicated by point D in Figure 2. We proceed in the same fashion as before, with the only difference that 1) we consider the function for application 1 to be $min(1000, x^3)$ and obviously the aggregated function becomes $min(1000, x^3) + x^2$; and 2) we are given a total allocation $\omega_g = 1180$ and utility sampling points 5 and 30;. We have the following available data: $W_{5,job1} = 25$, $W_{5,job2} = 125$, and so $\sum_m W_{5,m} = 150$; and $W_{30,job1} = 1000$ (constrained), $W_{30,job2} = 900$, and so $\sum_m W_{30,m} = 1900$. We calculate $ratio_g$ to be $(1180 - 150)/(1900 - 150) = 0.58$. With this value we estimate $\tilde{\omega}_{job1} = 640$ and $\tilde{\omega}_{job2} = 540$. This time the corresponding utilities for these job allocations are 8.6 for job 1 and 23.0 for job 2. As it can be observed, $\tilde{\omega}_{job1}$ and $\tilde{\omega}_{job2}$ satisfy equation 12, but they are far from each other. This is caused by the fact that application 1 is contributing differently to the aggregate function in the range of utilities 5 - 30, and under these circumstances $ratio_g$ is not accurate enough for our purpose of equalizing utilities if possible.

To overcome this problem we define app_ratio_m as:

$$app\_ratio_m = \frac{\frac{\sum_m W_{k+1,m}}{\sum_m W_{k,m}}}{\frac{W_{k+1,m}}{W_{k,m}}} \tag{14}$$

Notice that app_ratio_m observes how the difference between cells $W_{k,m}$ and $W_{k+1,m}$ is related to the overall system allocation that corresponds to $\sum_m W_{k,m}$ and $\sum_m W_{k,m}$. In the case that both $W_{k,m}$ and $W_{k+1,m}$ are constrained by the maximum achievable speed for application m, then we have that $app\_ratio_m = 0$.

We now define:

$$ratio_m = \begin{cases} ratio_g & W_{i,m} = u_{i,m}, i = k, k+1 \\ ratio_g * app\_ratio_m & \text{otherwise} \end{cases} \tag{15}$$

and interpolate the allocation $\tilde{\omega}_m$ that corresponds to application m given ω_g as $(W_{k+1,m} - W_{k,m}) * app\_ratio_m + W_{k,m}$.

Looking again to the example proposed in Figure 4, we get $app\_ratio_{job1} = (1900/150)/(1000-125) = 1.65$ and $app\_ratio_{job2} = (1900/150)/(900 - 25) = 0.33$. With these values, as well as the previously calculated $ratio_g$, we can calculate values $\tilde{\omega}_{job1} = 974$ and $\tilde{\omega}_{job2} = 195$, that still satisfy equation 12 but are closer in terms of utility. This technique has proven to work as expected for our input functions.

Once $\tilde{\omega}_m$ is correctly estimated, we can easily estimate the expected utility of application m as:

$$\tilde{u}_m(\omega_g) = \begin{cases} -\infty & \text{if } \omega_g = 0 \\ u_m(t_{\text{now}} + \frac{\alpha^{cr}_{N_m,m}}{\tilde{\omega}_m(\omega_g)}) & \text{otherwise} \end{cases} \tag{16}$$

Evolution of hypothetical utility over time. The hypothetical utility function assumes all jobs can be placed at once, which is usually not the case. This means that real placements differ from what is assumed by the hypothetical utility. In this case, when after time T the hypothetical utility is calculated again it may have a different form than the utility calculated at time t_{now}. Figure 5 illustrates two scenarios in which two applications compete for resources (only one of the applications can be placed at a time). In the upper chart, both applications have the same characteristics (maximum speed, importance level, submission time and completion time goal). In the upper chart, each application has different characteristics. The charts show how much CPU must be allocated to each application according to the calculated hypothetical utility as well as the aggregated allocation $(\omega_{m_1} + \omega_{m_2})$ necessary to achieve the utility of -0.1. Notice that when both applications have identical characteristics, the evolution of the required allocation for both applications to get the same hypothetical utility is complementary: the application that is placed and running presents a decreasing demand to get the same utility while the application that is stopped presents an increasing demand. The aggregated allocation keeps constant. In the second chart, where applications present different characteristics, the evolution of the allocation required by each applications differ. When the application placed has a tighter completion time goal, its requested allocation decreases more quickly than the demand for the other application increases. In this case, the aggregated required allocation for both applications to obtain identical hypothetical utility decreases slightly over time as the more constrained job progresses in execution.

4.4.2 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 jobs that are not placed, $\omega_m = 0$.

To calculate utility of application m given placement P that is calculated at time t_{now} for a control cycle that lasts time T, we calculate a hypothetical utility 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 this obtained hypothetical utility 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 job progress over its duration, and use 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 long-running work execution over time.

5 Experiments

In this section we present 5 experiments performed using a simulator already used and validated in [20] and [2].

Operation	**Cost**
Start VM	3.6s
Suspend VM	Memory demand * 0.0353s
Resume VM	Memory demand * 0.0333s
Live Migrate VM	Memory demand * 0.0132s

Table 1. Cost of virtualization operations

In Experiment One, we illustrate using a simple example how the hypothetical utility discussed in Section 4.4 guides the algorithm. In Experiment Two, we simulate a 25 node cluster to which a large number of identical jobs with identical deadline factors are submitted. In Experiment Three, we modify Experiment Two by randomly selecting (from three options) the deadline factor of each submitted job. In Experiment Four, we modify Experiment Two by randomly selecting (from three options) the execution time, maximum speed, and deadline factor for each submitted job. We then compare our algorithm against both FCFS and EDF. Finally, in Experiment Five, we modify Experiment Two by adding transactional workload to the system.

For the purpose of easily controlling the tightness of SLA goals, we introduce a relative goal factor which is define as a 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 all experiments, except for one (Experiment Four), the cost of virtualization operations (start, suspend, resume, migrate and move_and_resume) are considered. These costs were modeled using performance data obtained on our test systems running one of the most widely encountered virtualization products for Intel-compatible systems. These models show simple linear relationships between the VM memory footprint and the cost of the operation, as it can be seen in Table 1. Notice that the boot time observed for all our virtual machines was constant.

5.1 Experiment One: Hypothetical utility

In this experiment we illustrate how hypothetical utility (see Section 4.4) guides our algorithm to make placement decisions. We use three jobs, J1, J2, and J3 with properties shown in Table 2. We also use a single node with resource capacities shown in Table 2. 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 2. 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 an unrealistic control cycle $T = 1s$.

Figure 6 summarizes the scheduling decisions made by our algorithm in S1 (top) and S2 (bottom). Observe that the main difference between S1 and S2 occurs in control cycle 2: in S1, J1 is placed (and runs at full speed), whereas in S2, both J1 and J2 are placed with J1 running at half speed and J2 at full speed. The remainder of this section will describe the mechanisms that guide the algorithm to make these decisions.

Figures 7 and 8 show cycle-by-cycle executions of the algorithm for S1 and S2, respectively. Rectangular boxes show the outstanding work, $\alpha_m - \alpha_m^*$, work done, α_m^*,

Node	Memory	CPU speed
Capacity	2,000MB	1,000MHz

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

Table 2. Properties of Experiment One

value of hypothetical utility and corresponding CPU allocation for each job and various considered placement alternatives in subsequent control cycles. In most cycles in S1, only one placement is considered as the algorithm efficiently prunes the search space. Two alternative placements are considered in cycles 2 and 3. In cycle 2, we consider a placement, P1, that halves CPU allocation to J1 and starts J2 and a placement, P2, that leaves J1 running at full speed without starting J2. The same placement alternatives, P1 and P2, are considered in cycle 2 of S2. In S1, these two placements have the same hypothetical utility of 0.7 for both jobs. Since P1 and P2 have equal utilities, the algorithm opts to not make any changes and selects P2. In S2, due to the tighter completion time goal for J2, P2 has hypothetical utilities of 0.7 and 0.6 for J1 and J2 respectively, while P1 results in hypothetical utilities of 0.65 for both J1 and J2. Clearly, P1 is a better choice for S2.

The difference in hypothetical utilities of J2 in control cycle 2 between the two scenarios can be explained by looking at the maximum achievable utility of J2. 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 maximum achievable utility of 0.69 ($= (16 - 5)/16$), whereas in S2, it is only 0.58 ($= (12 - 5)/12$).

5.2 Experiment Two: Baseline

In this experiment, we examine the basic correctness of our algorithm by stressing it with a sequence of identical jobs, i.e., jobs with the same profiles and SLA goals. When jobs are identical, in the best scheduling strategy no placement changes (suspend, resume, migrate) should happen. This is the best possible behavior in this case, as no benefit to job completion times (when looked on as a vector) would 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 processors with properties shown in Table 3. To the system we submit 800 identical jobs with properties shown in Table 3. 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 600 s.

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 of the system mean that only three jobs will fit on a node at once. 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 ($2.7 * 17,600$), which is measured from the submission time.

Nodes	Memory	CPU Speed
Capacity	16,000MB	4x 3,900MHz

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

Table 3. Properties of Experiment Two

The maximum achievable utility for a job described in Table 3 is 0.63. This utility 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 need a 37% of the time between the submission time and the completion time goal to complete. This utility is an upper bound for the job, and will be decreased if queuing occurs.

Figure 9 shows the number of jobs in the system (already submitted), and the number of jobs that are placed at each moment in time. Note that the number of placed jobs never exceeds 75. In Figure 10, we show the average hypothetical utility over time as well as the actual utility achieved by jobs at completion time. When no jobs are queued, the hypothetical utility is 0.63 and it decreases as more jobs are delayed in the queue. Notice that the the utility achieved by jobs at completion time has the shape similar to that of the hypothetical utility, but is shifted in time by about 18000 sec. This is expected as that the hypothetical utility is predicting the actual utility that jobs will obtain at the time they complete, as thus is affected by job submissions, while the actual utility 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 done by the algorithm. Finally, Figure 11 shows the execution time for the algorithm at each control cycle when running on a 3.2GHz Intel Xeon node. It can be 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. In normal conditions, the algorithm produces a placement for this system in about 1.5s.

5.3 Experiment Three: Variable deadlines

In this experiment, we modify the conditions of Experiment Two (Section 5.2) by introducing three different relative goal factors. For each job, a relative goal factor is randomly chosen from three different possibilities – 1.9 (with a probability of 30%), 2.5 (with a probability of 40%), and 10 (with a probability of 40%). All jobs have the same execution characteristics as in Experiment Two.

Mixing jobs with different relative goal factors introduces a new range of options for improvement for managing the workload. Jobs with more relaxed goals can be suspended to permit newly submitted jobs with tighter goals to be started in their place. However, the longer a job with a relaxed goal is suspended, the more difficult its goal becomes to satisfy, making it comparable to a newly submitted job with a tight goal. Section 4.4 discusses in detail how the hypothetical utility guides the algorithm in the prediction of the achievable satisfaction for a job even when that job is currently not running.

Figure 12 shows the number of jobs in the system (already submitted), and the number of jobs that are placed at each moment in time. As in Experiment Two, we can never start more than 75 jobs simultaneously, owing to the memory constraints. Figure 13 shows the average hypothetical utility at each control cycle as well as the actual utility achieved by jobs at completion time, and the maximum achievable utility for jobs with relative goal factors 1.9, 2.5 and 10. Remember that all jobs have identical characteristics so their maximum achievable utility at the time they are submitted is the same for all jobs with the same relative goal factor. Note that the average hypothetical utility is no longer less than or equal to 0.63, as was the case in Experiment Two, as different deadline factors change the maximum achievable utility. However, the average hypothetical utility is still governed by the number of jobs in the system, and (in particular) the number of submitted jobs that are not currently placed (the job queue). The actual utility obtained by jobs at completion time is close to the maximum achievable utilities calculated for each relative goal factor. Our technique aims to equalize the utility at completion time for all jobs in the system, but in this scenario the presence of three different relative goal factors prevents it from achieving it – jobs with relative goal factor of 10 can achieve higher utility than the jobs with relative goal factor 1.9 without interfering. But notice that when the hypothetical utility decreases because some queueing is happening, less resources are allocated to the jobs with relative goal factor 10 in order to maintain as high as possible the utility achieved by jobs with tighter relative goal factors at completion time. This fact can be observed short after times 40,000s and 100,000s – the algorithm decides to sacrifice the utility of jobs with relative goal factor 10 to keep jobs with relative goal factors 2.5 and 10 close to their maximum achievable utility values. Even the number of jobs with relaxed relative goal factor completing is reduced at some of these periods, allowing other jobs to be run instead.

Figure 14 shows that the algorithm elects to both suspend and migrate jobs during the course of this experiment. While the load on the system is the same in this experiment and the previous one, in this case the multiple deadline factors mean that making placement changes after a job has been started is a useful way to improve the utility of the system (as can be seen by comparing Figures 10 and 13).

5.4 Experiment Four: Randomized jobs

In this section, we simulate the system exercised with jobs of various 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.

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.

In this experiment, we use eight different inter-arrival times, ranging in increments of 50s from 50s to 400s, and continue to submit jobs until 800 have completed. The experiment is repeated for the three mentioned algorithms: our algorithm (APC), EDF, and FCFS.

Figure 15 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.

Figure 16 shows the penalty for EDF's higher on-time completion rate at low inter-arrival times – 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 15, shows our algorithm's ability to making few changes to the system whilst still achieving a high on-time rate.

Figure 17 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 400, 300, 200, 100, and 50 seconds, in Figure 17 (a), (b), (c), (d), and (e), respectively. Points with distance to the goal greater than zero, indicate jobs that completed before their goal. Observe that for inter-arrival times of 200s or greater, 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 job length.

However, as the inter-arrival time becomes 100s or less, the algorithms produce different distributions of distance 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.

5.5 Experiment Five: Heterogeneity

The last of the experiments included here illustrates how our integrated management technique is applicable to combined management of transactional and long-running workloads.

We extend Experiment Two by adding transactional workload to the system. The experiment will show how our algorithm will allocate resources to both of the workloads in a such a way that equalizes their satisfaction in terms of distance between their actual response time and their response time goal. 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 5.2. 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 Two. This was done to ensure that the two different types of workload compete only for CPU resources.

Figure 18 shows the utility function used for the transactional workload. It shows how much CPU power must be allocated to this application for it to achieve a certain level of utility. The utility of transactional workloads is calculated as described in Section 3.3. A utility of zero means that the actual response time exactly meets the response time goal: lower utility values indicate that the response time is greater than the goal (the requests are being serviced too slowly), and higher utility values indicate that the response time is less than the goal (the requests are being serviced quickly). The maximum achievable utility 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.

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. In this state, the transactional application gets as much CPU power as it can consume, as there is little or no contention with long-running jobs. As more long-running jobs are submitted, following the workload properties described in Section 5.2, the hypothetical utility for those long-running jobs starts to decrease as the system becomes increasingly crowded. As soon as the hypothetical utility calculated for the long-running jobs becomes lower that the utility 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 utility each achieves is equalized. At the end of the experiment the job submission rate is slightly decreased, what results in more CPU power being returned to the transactional workload again. Figure 19 shows the utility for both of the workloads during the experiment. The utility for both workloads is continuously adjusted by dynamically allocating resources over time. Figure 20 shows the particular allocation at each moment of the experiment, as well as the CPU demand that would make each workload achieve its maximum utility. Notice how, as it was pursued, our technique makes an uneven distribution of resources in terms of CPU capacity, but it results in an even level of utility across the workloads.

6 Conclusions and future work

In this paper we present a technique that allows an integrated management of heterogeneous workloads, composed of transactional applications and long-running jobs, dynamically placing the workloads in such a way that equalizes their satisfaction. We use utility 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 as compared to other well-known scheduling algorithms. We carry on our experiments with a simulator already used and validated against a system prototype in [20, 2]. While here we mainly focus on the description and evaluation of the management of long-running jobs, transactional workloads were widely covered in [2]. We expect to extend this technique in the future to offer explicit support of the characteristics of parallel and distributed long-running jobs.

Acknowledgments

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] U. Balli and J. S. Anderson. Utility accrual real-time scheduling under variable cost functions. *IEEE Trans. Comput.*, 56(3):385–401, 2007. Member-Haisang Wu and Senior Member-Binoy Ravindran and Member-E. Douglas Jensen.

[2] D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguadé. 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.

[3] B. N. Chun and D. E. Culler. 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*, page 30, Washington, DC, USA, 2002. IEEE Computer Society.

[4] G. Czajkowski, M. Wegiel, L. Daynes, K. Palacz, M. Jordan, G. Skinner, and C. Bryce. Resource management for clusters of virtual machines. pages 382–389, Cardiff, UK, May 2005.

[5] D. M. David Vengerov, Lykomidis Mastroleon and N. Bambos. Adaptive data-aware utility-based scheduling in resource-constrained systems. Sun Technical Report TR-2007-164, Sun Microsystems, April 2007.

[6] S. Feizabadi and G. Back. Automatic memory management in utility accrual scheduling environments. In *ISORC '06: Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, pages 11–19, Washington, DC, USA, 2006. IEEE Computer Society.

[7] I. Foster, T. Freeman, K. Keahey, D. Scheftner, B. Sotomayor, , and X. Zhang. Virtual clusters for grid communities. Singapore, May 2006.

[8] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *IEEE Real-Time Systems Symposium*, pages 112–122, 1985.

[9] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic placement for clustered web applications. In *WWW Conference*, Edinburgh, Scotland, May 2006.

[10] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. 2004.

[11] T. Kimbrel, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic application placement under service and memory constraints. In *International Workshop on Efficient and Experimental Algorithms*, Santorini Island, Greece, May 2005.

[12] C. B. Lee and A. Snavely. On the user-scheduler dialogue: Studies of user-provided runtime estimates and utility functions. *Int. J. High Perform. Comput. Appl.*, 20(4):495–506, 2006.

[13] C. B. Lee and A. E. Snavely. 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*, pages 107–116, New York, NY, USA, 2007. ACM.

[14] B. Lin and P. Dinda. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proc. ACM/IEEE Supercomputing*, Seattle, WA, Nov. 2005.

[15] G. Pacifici, W. Segmuller, M. Spreitzer, M. Steinder, A. Tantawi, and A. Youssef. Managing the response time for multi-tiered web applications. Technical Report Tech. Rep. RC 23651, IBM, 2005.

[16] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi. Dynamic estimation of cpu demand of web traffic. In *VALUETOOLS*, Pisa, Italy, Oct. 2006.

[17] G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef. Performance management for cluster-based web services. *IEEE Journal on Selected Areas in Communications*, 23(12), Dec. 2005.

[18] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 289–302, New York, NY, USA, 2007. ACM.

[19] B. Ravindran, E. D. Jensen, and P. Li. 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)*, pages 55–60, Washington, DC, USA, 2005. IEEE Computer Society.

[20] M. Steinder, I. Whalley, D. Carrera, I. Gaweda, and D. Chess. Server virtualization in autonomic management of heterogeneous workloads. In *10th IEEE/IFIP Symposium on Integrated Management (IM 2007)*, Munich, Germany, 2007.

[21] Sun Microsystems. Behavior of mixed workloads consolidated using Solaris Resource Manager software. Technical report, May 2005.

[22] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proc. Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[23] X. Wang, D. Lan, G. Wang, X. Fang, M. Ye, Y. Chen, and Q. Wang. Appliance-based autonomic provisioning framework for virtualized outsourcing data center. In *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*, page 29, Washington, DC, USA, 2007. IEEE Computer Society.

[24] Z. Wang, X. Zhu, P. Padala, and S. Singhal. Capacity and performance overhead in dynamic resource allocation to virtual containers. *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, pages 149–158, May 21 2007-Yearly 25 2007.

[25] H. Wu, B. Ravindran, E. D. Jensen, and P. Li. Energy-efficient, utility accrual scheduling under resource constraints for mobile embedded systems. *Trans. on Embedded Computing Sys.*, 5(3):513–542, 2006.

[26] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif. On the use of fuzzy modeling in virtualized data center management. *Autonomic Computing, 2007. ICAC '07. Fourth International Conference on*, pages 25–25, 11-15 June 2007.

[27] X. Zhu, Z. Wang, and S. Singhal. Utility-driven workload management using nested control design. *American Control Conference, 2006*, pages 6 pp.–, 14-16 June 2006.

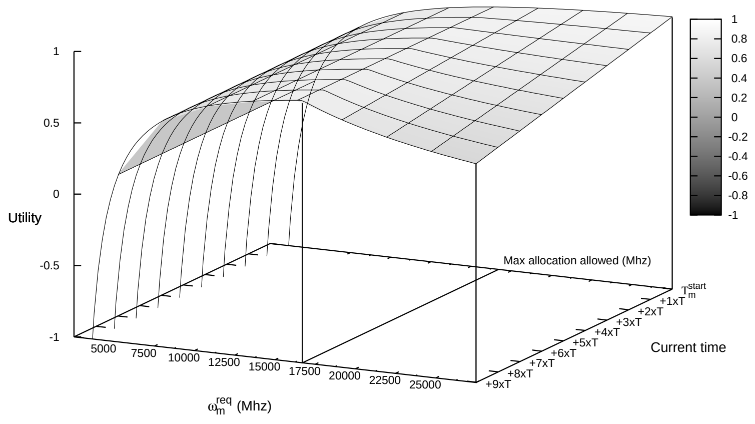


Figure 1. Evolution of maximum achievable utility for long-running jobs

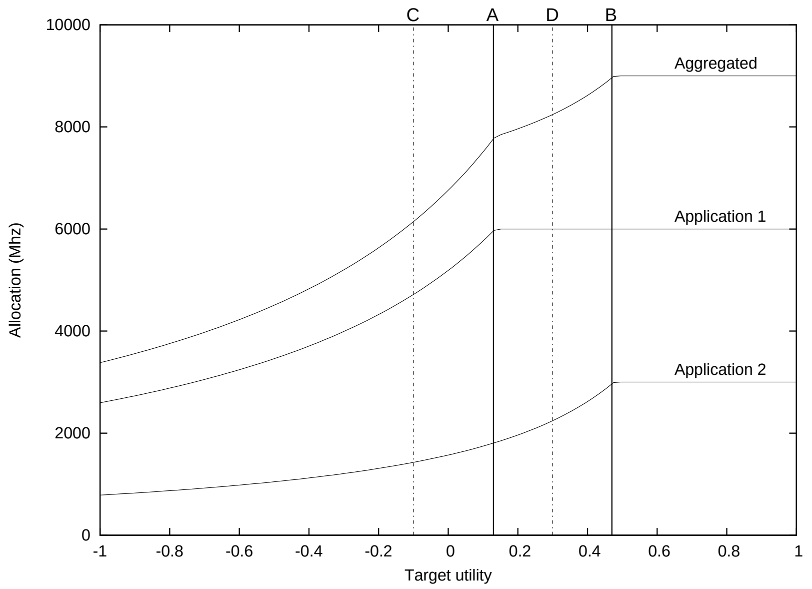


Figure 2. Allocation as a function of target utilities as used to estimate application utility given aggregate CPU allocation (used to fill vectors V and W). Notice that the maximum achievable utility for some jobs may be lower than the target utility (i.e. target utilities beyond point A for application 1, and beyond point B for application 2). Cells of vectors W and V for application 1 that correspond to target utilities beyond point A are filled with values 6000 and 0.14 (maximum achievable utility of 0.14 for allocation 6000Mhz). In the case of application 2 for target utilities beyond point B, vector cells are filled with values 3000 and 0.44

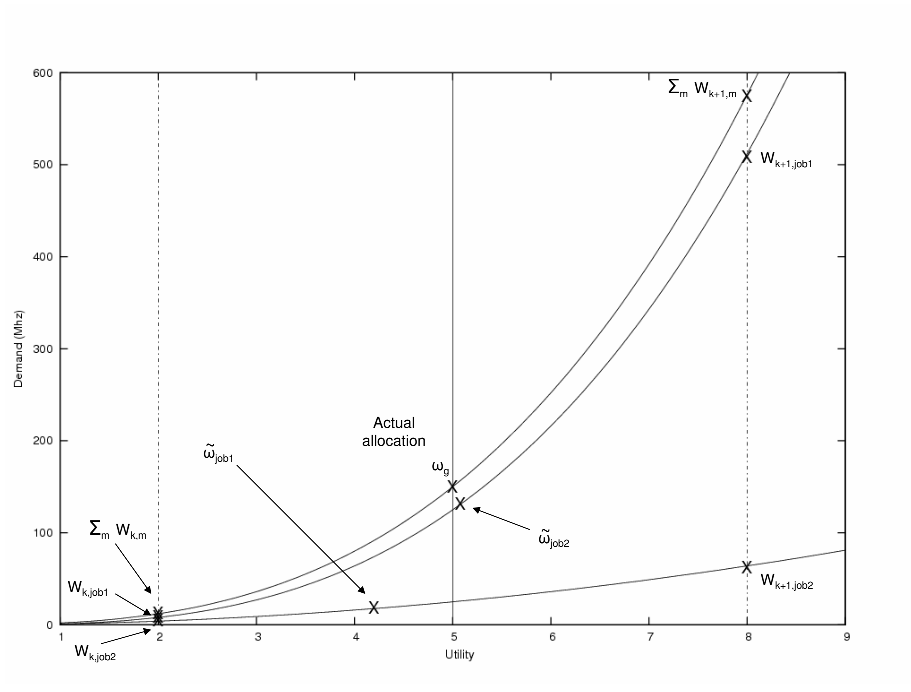


Figure 3. Estimating $\tilde{\omega}_m$ using $ratio_g$

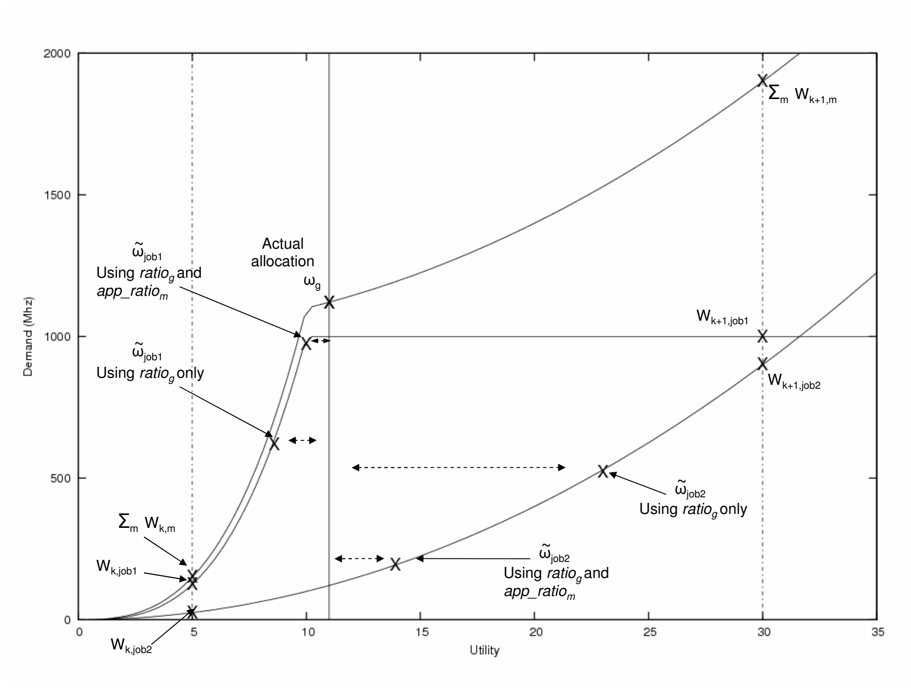


Figure 4. Estimating $\tilde{\omega}_m$ using $ratio_g$ and app_ratio_m

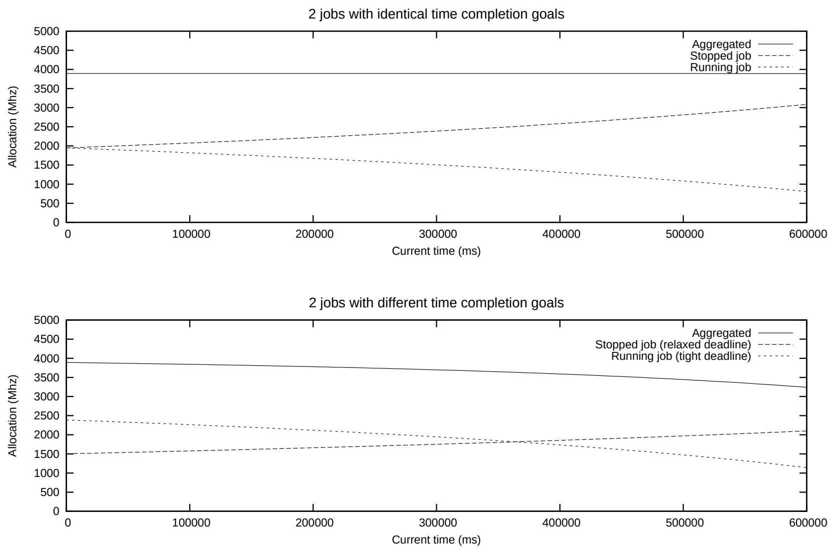


Figure 5. Hypothetical utility: effect of resource competition on the required allocation to obtain utility -0.1. Only one job can be placed (running) while the other is stopped. When both jobs have identical characteristics and same deadline (upper chart), the hypothetical utility for both jobs can be equalized over time by hypothetically allocating more CPU power to the job that is stopped to compensate the time it is stopped, resulting in a constant aggregate allocation. When both jobs are different (lower chart), their hypothetical utility still can be equalized but the aggregate allocation is not constant

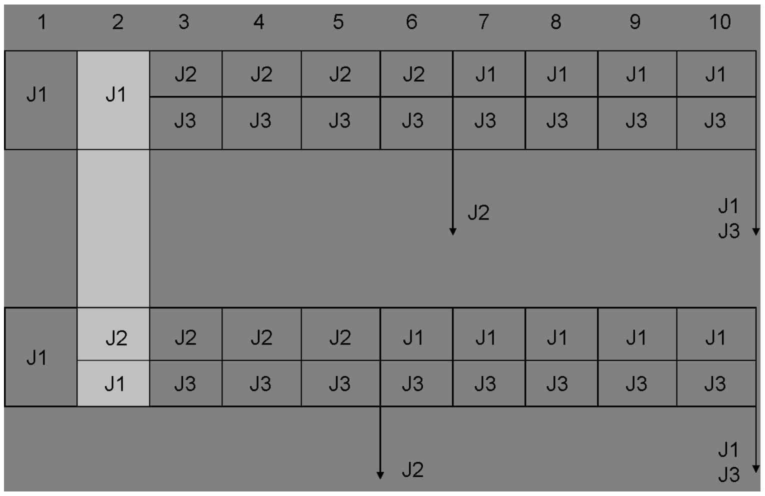


Figure 6. Experiment One: Summary

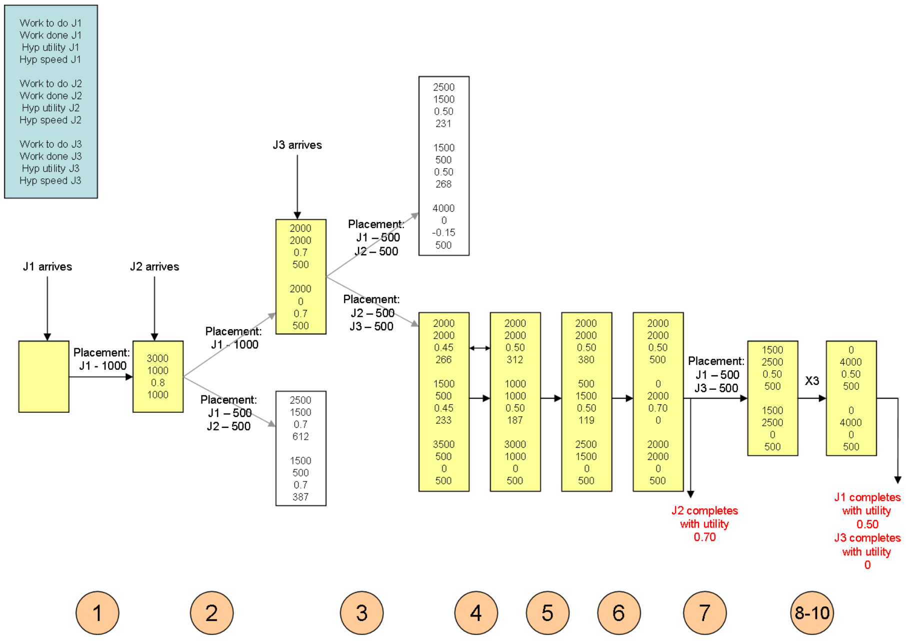


Figure 7. Experiment One: Scenario 1

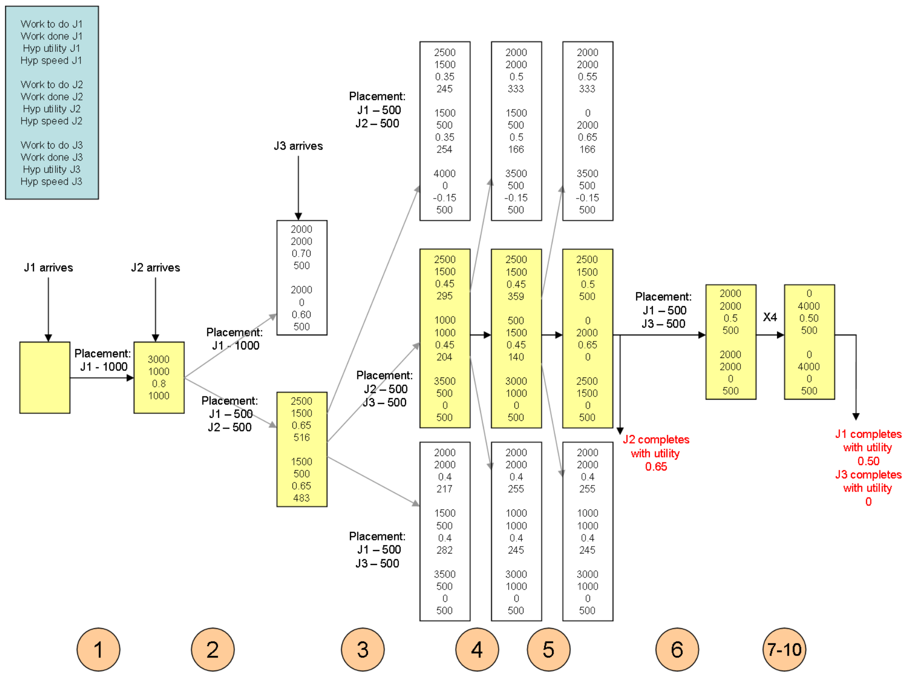


Figure 8. Experiment One: Scenario 2

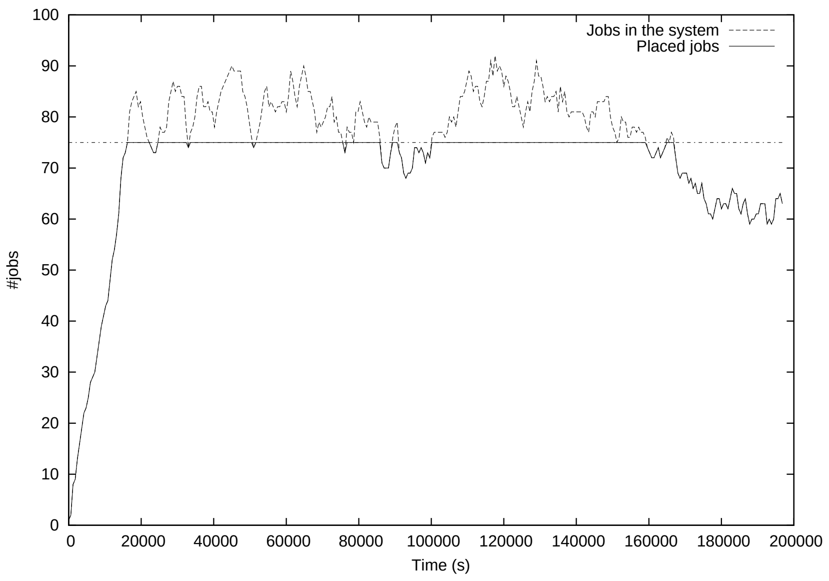


Figure 9. Experiment Two: Jobs in the system and jobs placed

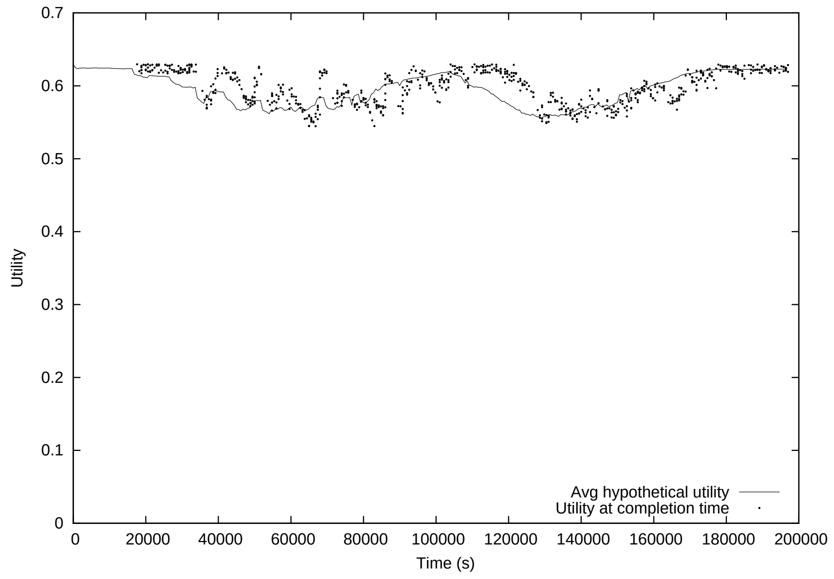


Figure 10. Experiment Two: Average hypothetical utility over time and actual utility achieved at completion time

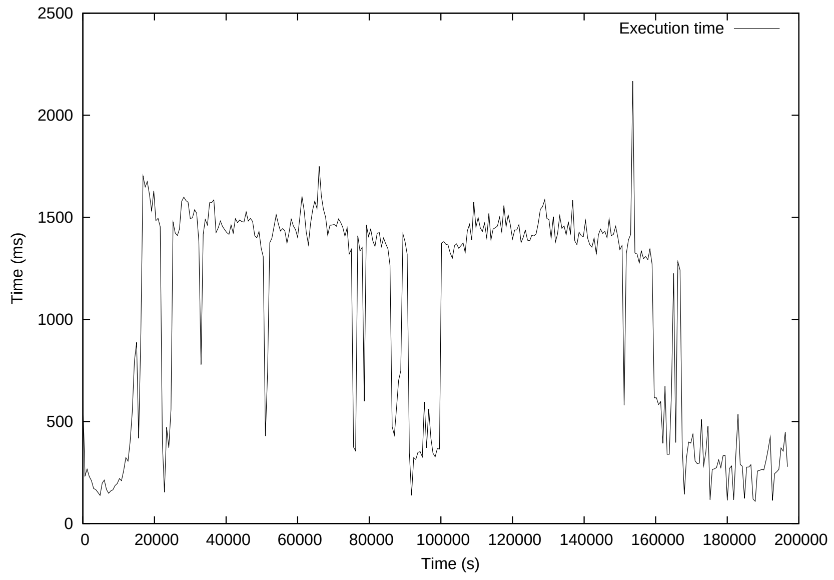


Figure 11. Experiment Two: Algorithm execution time

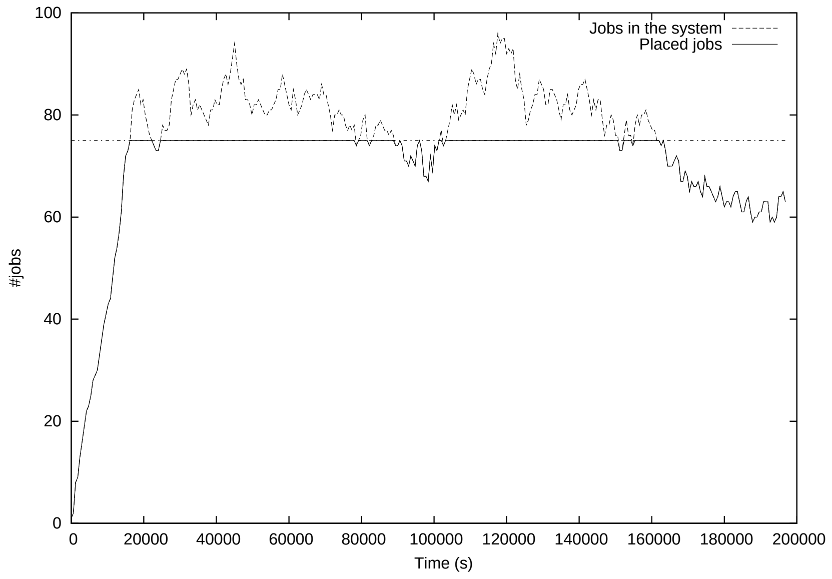


Figure 12. Experiment Three: jobs in the system and jobs placed

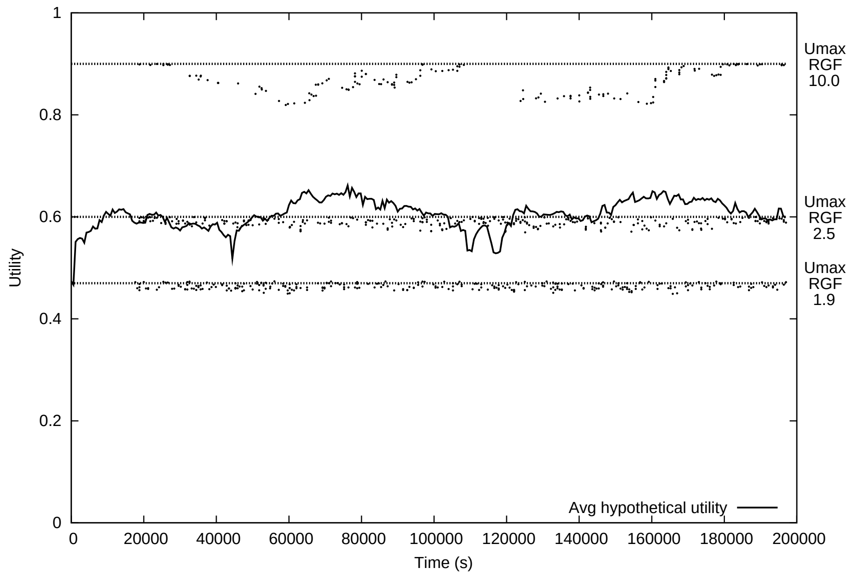


Figure 13. Experiment Three: average hypothetical utility over time and actual utility achieved at completion time

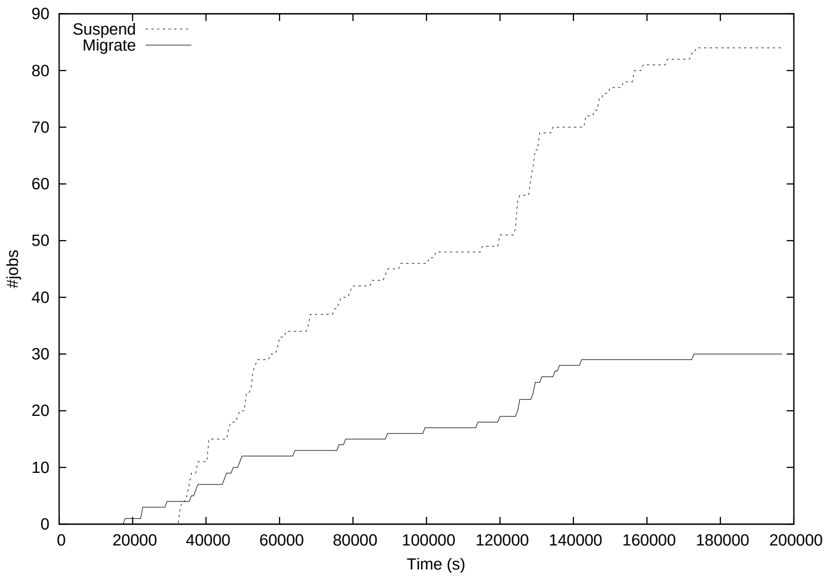


Figure 14. Experiment Three: total number of virtualization operations over time

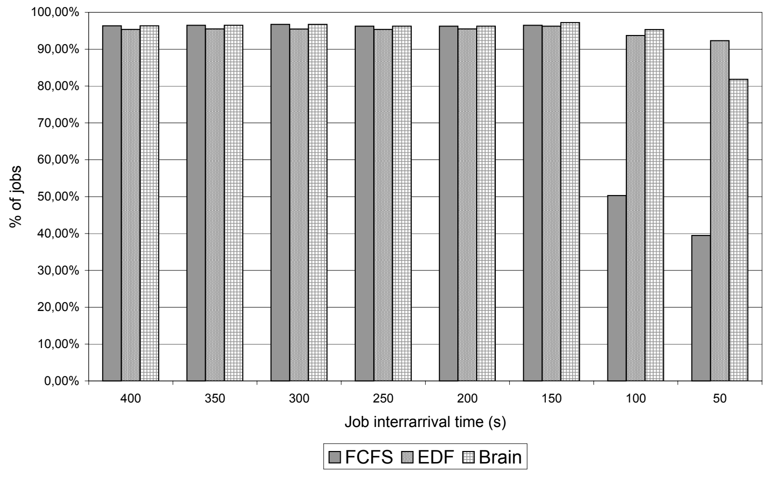


Figure 15. Experiment Four: Percentage of jobs that met the deadline

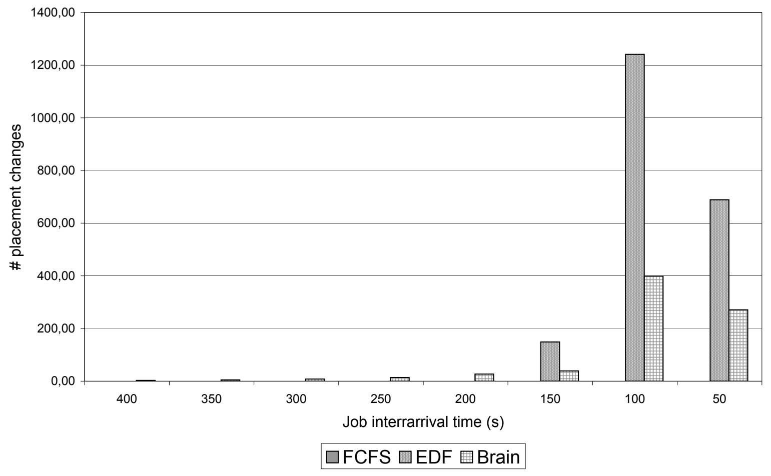
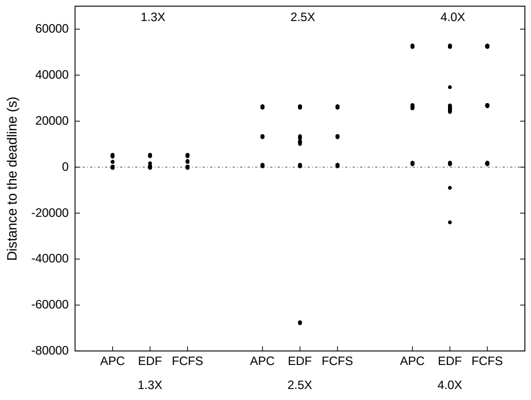
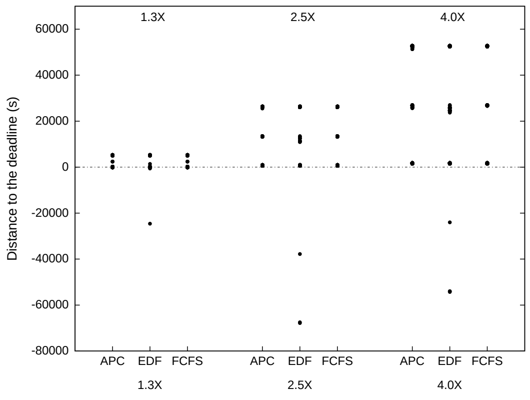


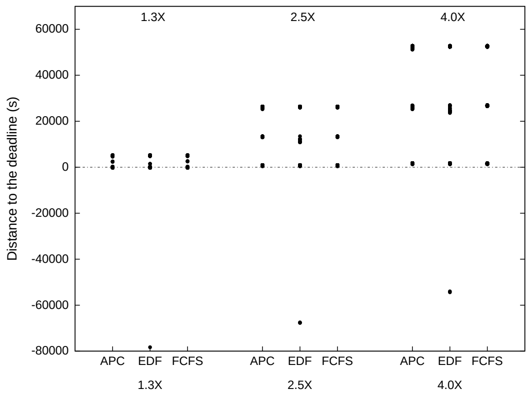
Figure 16. Experiment Four: Number of jobs migrated, suspended, and moved_and_resumed



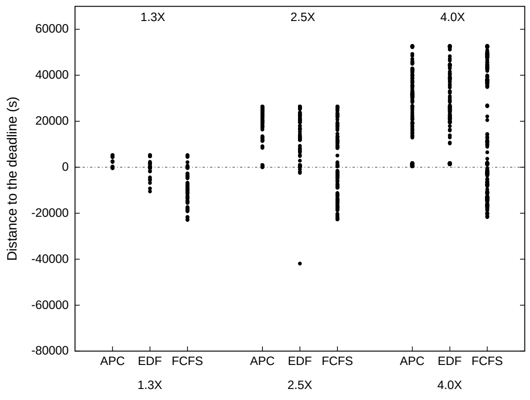
(a) 400s



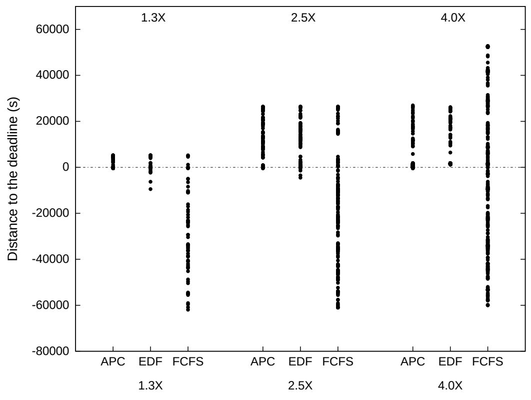
(b) 300s



(c) 200s



(d) 100s



(e) 50s

Figure 17. Experiment Four: distribution of distance to the goal at job completion time, for five different mean interarrival times (50s to 400s)

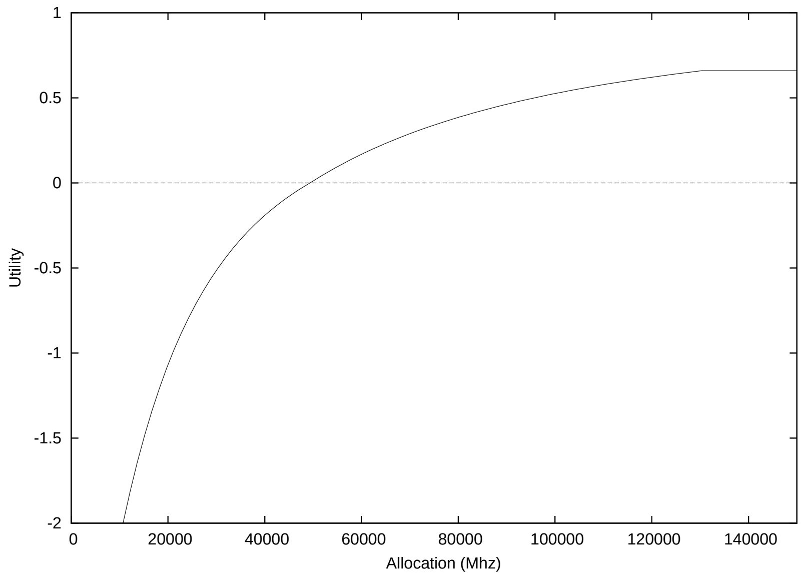


Figure 18. Experiment Five: utility function for the transactional workload (utility as a function of allocated CPU power)

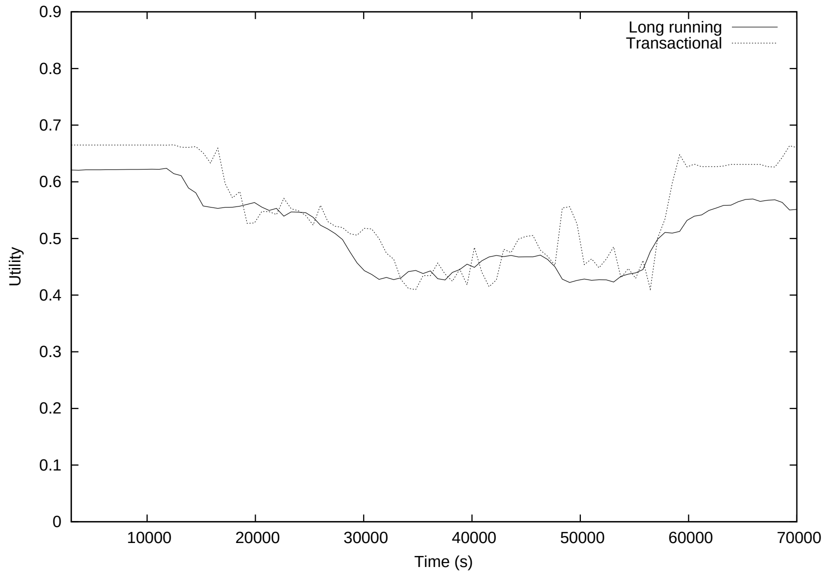


Figure 19. Experiment Five: actual utility for the transactional workload and average calculated hypothetical utility for the long-running workload

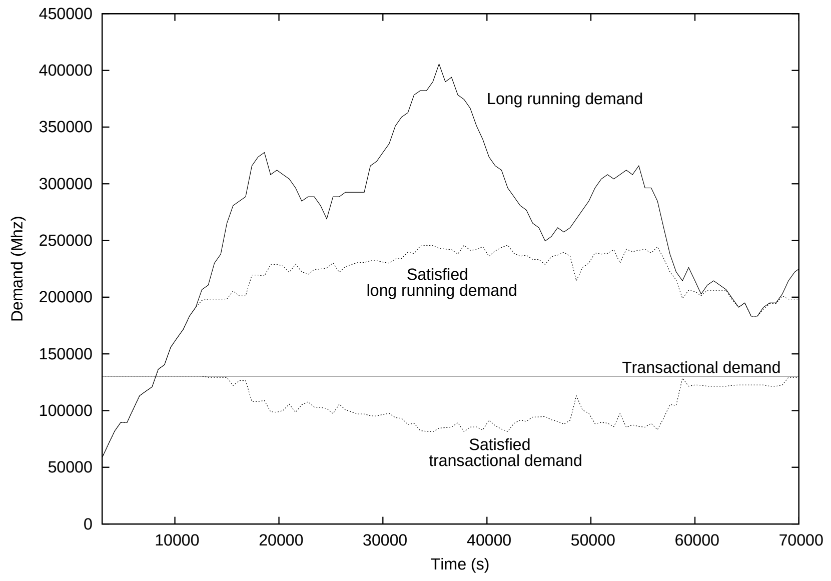


Figure 20. Experiment Five: CPU power allocated to each workload and CPU demands to achieve maximum utility