Configuring Workload Manager Control Parameters for Resource Pools

Jerome Rolia
Hewlett-Packard Labs
Palo Alto, CA, USA, 94302
Email: Jerry.Rolia@hp.com

Ludmila Cherkasova
Hewlett-Packard Labs
Palo Alto, CA, USA, 94302
Email: Lucy.Cherkasova@hp.com

Clifford McCarthy
Hewlett-Packard MSL
Richardson, TX, USA, 75080
Email: Clifford.McCarthy@hp.com

***Abstract*— Resource pools are computing environments that offer virtualized access to shared resources. When used effectively they can align the use of capacity with business needs (flexibility), lower infrastructure costs (via resource sharing), and lower operating costs (via automation). Using resources effectively can rely on a combination of workload placement and workload management technologies. Workload placement decides which workloads will share resources. Workload management governs short term access to resource capacity. It provides performance isolation within resource pools to ensure resource sharing even under high loads. A workload manager can have a direct impact both on an application's overall resource access quality of service and on the number of workloads that can be assigned to a pool. In this paper we take a detailed look at an application workload's demands. We explore tradeoffs in resource access quality of service received by the application and the minimum allocation of resources for the workload. We show that by careful selection of workload scheduling parameters along with a proposed fast allocation policy we can sometimes more than triple the number of workloads that can be assigned to a pool without sacrificing application workload quality of service or the efficiency of the resource pool.**

I. Introduction

Resource pools are collections of resources, such as clusters of servers or racks of blades, that offer shared access to computing capacity. Virtualization services offer interfaces that support the lifecycle management (e.g., create, destroy, move, size capacity) of resource containers (e.g., virtual machines, virtual disks) that are provided with access to shares of resource capacity (e.g., cpu, memory, input-output). This paper considers the efficient use of resources in such pools.

We assume that when managing such pools that application workloads are assigned to resource containers that are then associated with resources in the pool. Management occurs at several different timescales. Long term management corresponds to capacity planning and takes place over many months. Over a medium timescale, e.g., days or months, groups of resource containers are found that are expected to share resources well. These containers are then assigned to their corresponding resources. Capacity management tools can be used to automate such a process. For example, our capacity management tool takes into account detailed workload interactions and the overbooking of resources via statistical multiplexing [1] to automatically decide which workloads should share resources. Once resource containers are assigned

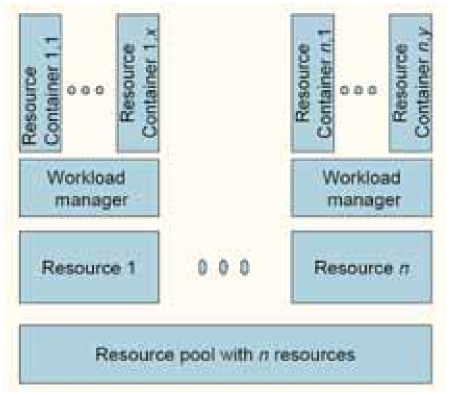

Fig. 1. Relationship between resource containers, workload managers, resources, and a resource pool.

to a resource, a workload manager for the resource governs access to resource capacity over short timescales, e.g., 15 seconds. A workload manager can provide static allocations of capacity or change the per resource container allocations based on time-varying workload demand. Figure 1 shows the relationship between resource containers, workload managers, resources, and a resource pool.

This paper considers the issue of choosing per resource container values for workload manager control parameters that ensure the efficient use of resources while providing adequate application resource access quality of service. Our work focuses on a proportional share scheduler approach towards workload management. Each resource container is pre-allocated specific shares of capacity for short time periods, e.g., 15 seconds. Then, based on the demands of the containers and the availability of resources, the allocations may be adjusted to ensure that each container gets the capacity it needs.

We consider a particular workload and determine the workload manager control parameters that affect its resource usage within its container most. We found that the careful selection of workload scheduling parameters along with an aggressive policy towards allocating resources to workloads with increasing demands makes it possible in some cases to more than triple the number of workloads that can be assigned to a pool without sacrificing application workload quality of service or the efficiency of per-workload resource usage. We believe this contributes to the more effective use of resource pools and in that way enables their benefits to be realized.

The workload manager we consider is described in Section II. Section III explains the metrics we used to characterize capacity usage and resource access quality of service. Section IV describes the workload we consider in this paper. The impact of workload manager control parameter settings on the metrics for the workload is presented in Section V. To improve upon these metrics we introduce a novel fast allocation control policy for the workload manager in Section VI and repeat the sensitivity analysis. Related work and conclusions are offered in Sections VII and VIII.

II. Workload Manager

We assume that each resource container is associated with an entitled number of shares of CPU resources. The entitled number of shares correspond to an upper bound on the allocation of resources for the container. A workload manager dynamically varies the allocation to permit a more efficient use of the resource pool. The workload manager we consider is layered upon a proportional share scheduler.

The proportional share scheduler we consider provides bounded access to resources for each resource container. The scheduler is configured to offer 10 msec CPU timeslices. Figure 2 shows a pie chart that illustrates a schedule for the scheduler that supports several resource containers. Each piece of the pie represents one timeslice. Similarly shaded slices correspond to the same resource container. With bounded access, the scheduler advances from slice to slice every 10 msec regardless of whether resources are used or not, i.e., it is non-work-conserving scheduler. This provides for performance isolation. Each container receives its particular service rate regardless of whether any of the containers are using resources. Such isolation can be desirable in a shared resource environment for enterprise applications as it gives the appearance of dedicated access to resources. Adding new workloads to the pool has little impact on the performance behavior of workloads already in the pool.

The schedule is chosen to provide each resource container with access to its allocated shares. The schedule spreads each resource container's shares as evenly as possible over the pie to deliver what may be perceived as a smooth service rate. Though the pie illustrates a schedule for access to one CPU, a workload may be assigned shares from many CPUs such that its total allocation equals its number of shares. Workload schedulers may use heuristics to best match the offered concurrency over all CPUs with the level of concurrency a workload is able to exploit. However, when more CPUs are used the service rate per-CPU diminishes and the per-CPU schedules may differ. We make no assumption about whether the schedules for multiple CPUs are synchronized.

An inherent problem of a fixed schedule, i.e., fixed allocations, is that resources may not be used as efficiently as desired. Each resource container must be sized to support its peak capacity requirements. Yet, most applications rarely need

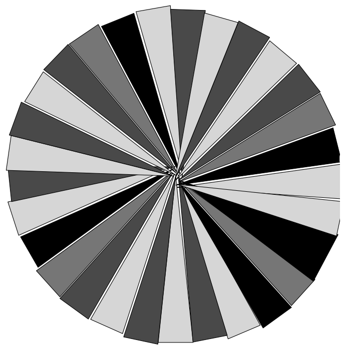

Fig. 2. Resource Allocation for A Proportional Share Scheduler

their peak amount. Workload managers aim to dynamically allocate the capacity associated with resource containers to facilitate sharing.

The workload manager we consider is layered on top of the proportional share scheduler and dynamically adjusts the allocation of each resource container based upon the current demands of the resource container. For example, if a workload goes idle, then its allocation can be reduced. If it becomes very busy it can be increased.

Adjusting the allocation of resources to containers based on demand permits statistical multiplexing within a resource pool [1]. We do not consider such multiplexing in this paper. Here, our analysis assumes that each workload gets access to capacity according to its demands and the allocation decisions of the workload manager algorithm. Our analysis is with respect to a workload within its resource container.

The workload manager we consider corresponds to a negative feedback controller. It has several control parameters. These include:

- schedule interval (this parameter only is for all workloads). The workload manager computes a new schedule for the proportional share scheduler at the end of each schedule interval;
- *gain* –a parameter that affects how quickly a workload's allocation increases or decreases based on its current demand;
- *minCPU allocation* – a minimum allocation amount, i.e., even in the absence of demand, the allocation can not go lower than *minCPU* amount;
- *maxCPU allocation* – maximum allocation amount for the workload;
- *lowerAllocUtil threshold* – a parameter that triggers a decrease of the allocation, i.e., if the measured utilization of allocation for a workload for the previous schedule interval drops below the *lowerAllocUtil* value, then the allocation is decreased based on the *gain* value, but it never goes below the *minCPU allocation* amount;

- *upperAllocUtil threshold* – a parameter that triggers an increase of the allocation, i.e., if a utilization of allocation goes above the *upperAllocUtil* then the allocation is increased based on the *gain* value, but it can not go higher than *maxCPU allocation* amount.

The allocation does not change when utilization of allocation falls within the range *lowerAllocUtil* and *upperAllocUtil*, and the allocation never goes out of the range *minCPU* allocation and *maxCPU* allocation. These conditions help to avoid hysteresis, limit the impact of one workload on others, and ensure resource access quality of service when a workload is transitioning from an idle to busy period. Note that: $0 \leq lowerAllocUtil \leq upperAllocUtil \leq 1$.

To describe the controller's algorithm in a more formal way, we use the following notation:

- i – the current time interval;
- D_i^{new} – a new incoming workload demand in interval i;
- $D_i^{carry\_forw}$ – the portion of demand that was not satisfied in interval $i-1$ and is therefore carried forward to interval i. For each time interval, it is computed according to the following formula:
$$D_i^{carry\_forw} = max(D_{i-1} - A_{i-1}, 0);$$
- D_i – the total demand in interval i,
$$D_i = D_i^{new} + D_i^{carry\_forw};$$
- A_i – the allocation provided by the controller in time interval i.

At the end of interval i, the workload manager computes a new allocation A_{i+1} for the workload for interval $i+1$ using the following policies:

1) If $lowerAllocUtil \leq D_i/A_i \leq upperAllocUtil$ then there is no change in the allocation, and $A_{i+1} = A_i$.
2) If $D_i/A_i \leq lowerAllocUtil$ then the controller attempts to decrease the next interval allocation:
$$A_{i+1} = A_i - gain \times (A_i - \frac{D_i}{lowerAllocUtil}).$$
If $A_{i+1} \leq minCPU$ then $A_{i+1} = minCPU$.
3) If $upperAllocUtil \leq D_i/A_i$ then the controller attempts to increase the next step allocation:
$$A_{i+1} = A_i + gain \times (\frac{D_i}{upperAllocUtil} - A_i).$$
If $maxCPU \leq A_{i+1}$ then $A_{i+1} = maxCPU$.

The workload manager takes the computed allocations for each workload's resource container and associates them with a schedule, i.e., for time interval $i+1$. The proportional share scheduler then serves short time slices according to the schedule until the end of the interval.

Since allocation can not go lower than *minCPU* amount it may be tempting to set such an allocation to a very small value. However, in this case, it may take several schedule intervals to increase an allocation to a correct size when there is a burst of incoming demand. This may present a resource access quality of service issue for interactive workloads with infrequent bursts of requests as the requests that start the bursts may incur long response times.

The choice of lower and upper utilization of allocation thresholds is based on the responsiveness requirements and arrival process of a workload. The greater the burstiness in arrivals, and the greater the responsiveness requirements, the lower the acceptable utilization of allocation. This is because utilization of allocation is measured over an interval, e.g., 15 seconds, so it conceals the bursts of demand within the interval. For this reason resource allocations are typically larger than resource demands. This limits the potential utilization of resources in a resource pool.

III. METRICS

We now define the metrics for measuring capacity usage and resource access quality of service. These are operational measures that are computed over a representative time period for a workload's demands, e.g., hours or months. We also introduce the notion of demand variation. It is a measure of workload burstiness that quantifies the impact of changing demands on the workload manager control algorithm.

The original workload and its demands are characterized as a trace of CPU demand values for the time period, with one CPU demand value per schedule interval. We compute values for the metrics with respect to a particular set of workload manager control parameters by replaying the trace through the workload manager control algorithm.

To facilitate the comparison of alternative parameter settings we normalize the metrics with respect to a *reference system*. The reference system is assumed to have a fixed capacity, i.e., its allocation does not change over time. In a reference system with N CPUs, all N CPUs are statically allocated to workload for all time intervals. We denote this reference capacity as allocation A_{ref}.

Consider a trace of demands D_i^{new} for intervals $i = 1, ..., T$ for a trace with T demand measurements. Let A_i be the CPU allocation for interval i as it is computed by the workload manager control algorithm.

We define *capacity usage U* as a workload's average allocation with respect to the reference system's capacity:
$$U = \frac{(\sum_{i=1}^{i=T} A_i)/T}{A_{ref}}.$$

A lower value for U corresponds to a lower usage of the reference capacity and hence leads to greater efficiency because the unused resources can be used by other workloads.

We define resource access quality of service (QoS) using the metric *QoS satisfied demand* D^{QoS}. This is the portion of total demand that is satisfied in intervals that have utilization of allocation less than or equal to *upperAllocUtil*. We define D^{QoS} as follows:

$$D^{QoS} = \sum_{i=1}^{i=T} D_i, \text{ such that } \frac{D_i}{A_i} \leq upperAllocUtil.$$

Note that D_i includes carry forward demand $D_i^{carry\_forw}$, since $D_i = D_i^{new} + D_i^{carry\_forw}$ as defined in Section II. The quicker the workload manager controller adapts to provide the correct allocation the higher fraction of the carried forward demand might be QoS satisfied demand.

Intuitively, the reference system helps to set the QoS expectations. Since its allocation does not change over time and it always allocates the maximum capacity of the reference system, it shows what fraction of workload demands can be QoS satisfied under its maximum possible capacity allocation.

To facilitate comparison with the reference system (and in such a way, between different systems), we normalize the QoS satisfied demand that corresponds to a workload control parameter configuration scenario, D^{QoS}, with respect to the QoS satisfied demand of the reference system, D_{ref}^{QoS}. This normalization characterizes the portion of demand that is QoS satisfied demand with respect to the reference system. We refer to this as *normalized QoS satisfied demand* Q^D. It is defined as:

$$Q^D = \frac{D^{QoS}}{D_{ref}^{QoS}}.$$

When $Q^D = 1$ the amount of QoS satisfied demand is the same as was measured for the reference system. For systems with higher capacity than the reference system, Q^D can be greater than 1, meaning that the amount of QoS satisfied demand is higher than for the reference system.

We also define the metric *Satisfied Demand* S^D. This is the portion of total demand that is satisfied in intervals where the demand is less than or equal to the allocation:

$$S^D = \frac{\sum_{i=0}^{i=T} D_i}{D} \text{ such that } D_i \leq A_i,$$

where D is the sum of demands D_i^{new} over the T time intervals.

For metrics Q^D and S^D we keep track of the percentage of the time intervals that satisfy the respective requirements. We name these as Q^P and S^P. These metrics bring additional perception of time for characterizing resource access quality of service. For example, the Q^P metric reflects the percentage of time intervals (amount of time) during which QoS satisfied access to CPU capacity is provided. These metrics have values between 0 and 1. The values are not normalized.

We rely on resource usage based metrics as measures of application workload quality of service because metrics from within an application's context, such as response times, are hard to obtain in general. We use the metrics to show how well we are providing capacity in proportion to a workload's needs.

In some sense, the Q^D value reflects how quickly the workload manager reacts to changing loads. When $Q^D < 1$ then a lower percentage of demand is being satisfied in the intervals with utilization of allocation less than *upperAllocUtil* than for the reference system. When $Q^D > 1$ it suggests that more demand is being satisfied in such intervals than for the reference system.

Finally, we introduce the notion of demand variation. This value helps to characterize the impact of workload burstiness on the workload manager control algorithm's computed sequence of allocations, i.e., $A_1, ..., A_T$. We define demand variation for interval i as:

$$demand\ variation_i = \frac{max(minCPU, D_i)}{max(minCPU, D_{i-1})}.$$

It is the ratio of successive demands as observed by the workload manager control algorithm to decide whether the allocation for interval $i{+}1$ needs to be increased. If the allocation must be increased then by definition there will be an impact on Q^D. In this way demand variation helps to understand the limit on the value for Q^D for a particular workload and workload manager control parameter settings. For a specific utilization of allocation range $(lowerAllocUtil, upperAllocUtil)$, we have an upper bound on the limit on demand variation of:

$$demand\ variation = \frac{upperAllocUtil}{lowerAllocUtil}.$$

If successive utilization of allocation values tend towards the midpoint of the range then the limit will be lower, e.g.:

$$demand\ variation = \frac{(upperAllocUtil + lowerAllocUtil)/2}{lowerAllocUtil}.$$

Later, we illustrate what portion of demand variations are less than or equal to such limits for various workload controller parameter settings.

IV. File server workload

For the purpose of our study we obtained a trace of CPU demands for a file server from a software development environment that is to be hosted in a resource pool. This is our reference system. The trace was recorded over 140 days and includes a measurement of CPU utilization every 15 seconds. The file server had fixed access to 3 CPUs and was not governed by a workload manager.

Figures 3 and 4 illustrate the entire 140 days of CPU demands in the trace and an arbitrarily chosen day, respectively. From a CPU utilization perspective, Figure 3 shows that 3 CPUs appear to be adequate for this reference system. From Figure 4 we see that demands are present throughout the day with frequent changes in resource requirements.

Figures 5 and 6 give the Cumulative Distribution Function (CDF) for the CPU demands for the full trace, and a CDF that shows how much of the total demand D is incurred in intervals with the greatest demands. The later curve is obtained by sorting the per-interval demands from largest to smallest.

Figure 5 shows that 90% of demands are less than 0.4 CPU and that 95% of demands are less than 0.6 CPU. Figure 6

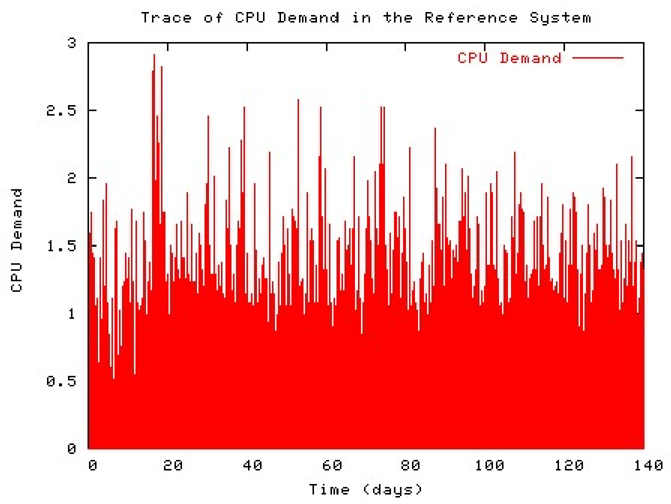

Fig. 3. 140 Days of CPU Demand Data for a File Server

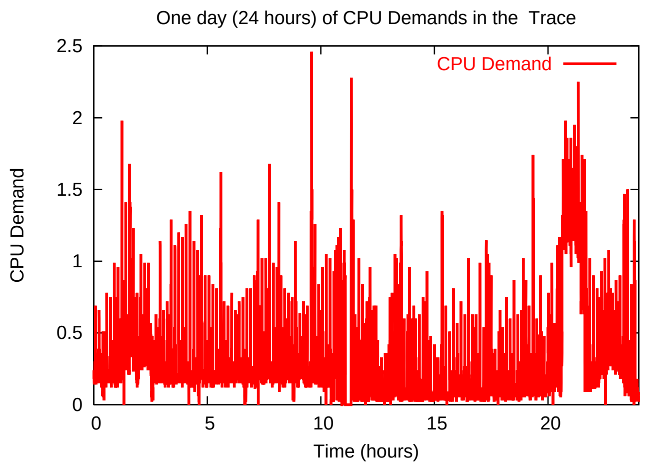

Fig. 4. CPU Demand Data for Day 17

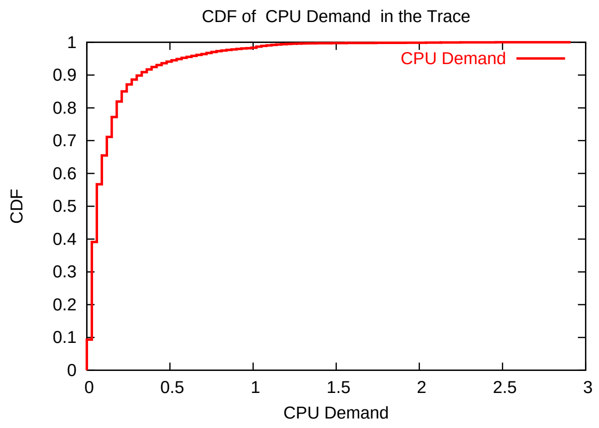

Fig. 5. CDF of Demands for Full Trace

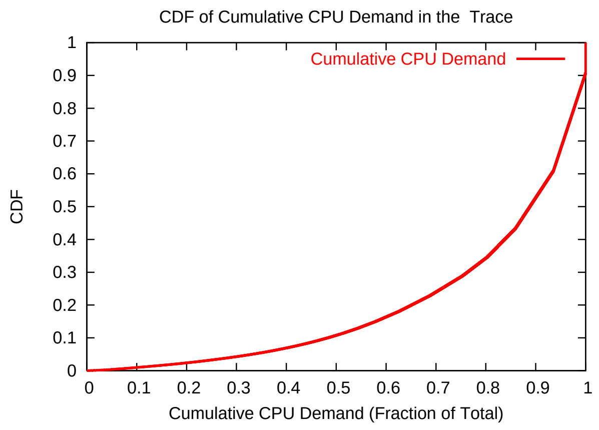

Fig. 6. Relationship between CPU Demand and Intervals for Full Trace

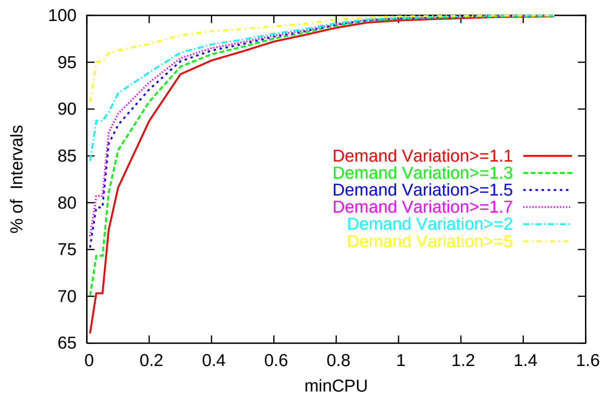

Fig. 7. *minCPU* and Demand Variation v.s. Percentage of Intervals

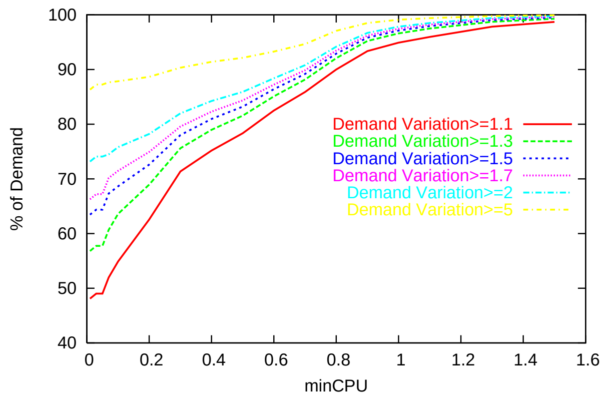

Fig. 8. *minCPU* and Demand Variation v.s. Percentage of Total Demand

shows that 1% of the intervals with highest demand, i.e., 0 to 0.01 on the y-axis, contribute approximately 10% of overall CPU demand in the trace, i.e., 0 to 0.1 on the x-axis. Furthermore, 10% of the intervals with lightest load, i.e., 0.9 to 1.0 on the y-axis, contribute to only 1% of the overall demand in the trace, i.e., 0.99 to 1.0 on the x-axis. This is a bursty workload.

Figures 7 and 8 show the percentage of intervals and demands that fall in intervals with demand variation less than specific thresholds, respectively. Each figure has several curves, one per threshold.

Consider demand variation limits of 1.3 and 1.1. Figure 7 shows that with a *minCPU=0.6* approximately 3% or 4% of time intervals will cause reallocations and negatively impact Q^D. Figure 8 shows that these points correspond to roughly 20% of the total demand. A *minCPU=0.8* has only 2% of the points causing reallocations and affects approximately 14% of the demand. From Figure 6, the top 1% of points correspond to the top 10% of demands. Figure 8 shows that even $minCPU = 1.5$ has 2% of total demand that will be affected by reallocations.

Resource sharing will always have an impact on application QoS. In general, we aim to find a trade-off that provides acceptable application QoS and makes efficient use of resources, i.e., economically sound based on business need.

V. Sensitivity Analysis of Workload Manager Control Parameters

This section considers various workload manager control parameter settings for the file server workload and their impact on Q^D, U, Q^P and S^P. We consider three workload quality of service scenarios: high, medium, and low. These correspond to utilization of allocation ranges *(lowerAllocUtil,upperAllocUtil)* of (0.233,0.433), (0.4, 0.6), and (0.566, 0.766), with mean utilization of allocation goals of 0.333, 0.5, and 0.666, respectively.

For each QoS scenario we explore the impact of different values for the *gain*, *minCPU* and *maxCPU* parameters on resource usage efficiency and resource access quality of service metrics.

A. Sensitivity to Gain and minCPU Parameters

Gain is the parameter in the workload manager control algorithm that affects how quickly a workload's allocation increases or decreases based on its current demand. *minCPU* defines the minimum share of CPU that is always allocated to a workload independent on its CPU demand: it means that at any point in time, the scheduler will allocate at least *minCPU* amount to the workload whether it needs CPU resources or not.

Figures 9 through 11 show the relationship between the metrics QoS satisfied demand Q^D and CPU capacity usage U with the different settings of *minCPU* for the high, medium, and low QoS target scenarios, respectively. Each figure presents three Q^D curves that correspond to values of *gain*=0.5, 1, and 2, respectively. Each figure also includes three curves showing the capacity usage U for the corresponding values of *gain*.

Figure 9 shows that *gain* has very little impact on capacity usage metric U. The U curves have values that are practically identical. The same is true for the medium and low QoS scenarios. The *gain* parameter does however have some impact on values of QoS satisfied demand Q^D. For lower *minCPU* levels, higher values for the *gain* parameter provide better values for Q^D. However, for higher values of QoS satisfied demand Q^D, e.g. 80%, higher values of *minCPU* are required. For these higher *minCPU* values, the impact of the *gain* parameter on Q^D is practically indistinguishable.

We conclude that a *gain = 1* is a reasonable choice for this workload and use it in our remaining analyses. We do not consider the *gain* parameter further in this study.

These figures also show the relationship between high, medium, and low QoS scenarios with capacity usage U. When *minCPU = 1*, capacity usage U is 35%, 33%, and 32% respectively. This metric tells us that for this value of *minCPU* parameter the three scenarios have roughly the same

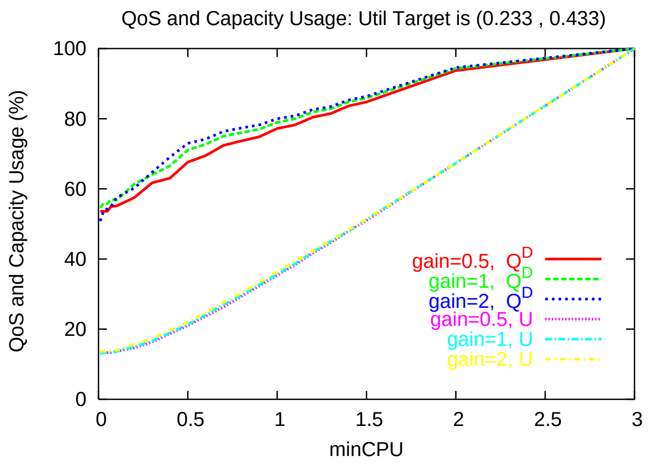

Fig. 9. Impact of Gain Parameter for High QoS Case

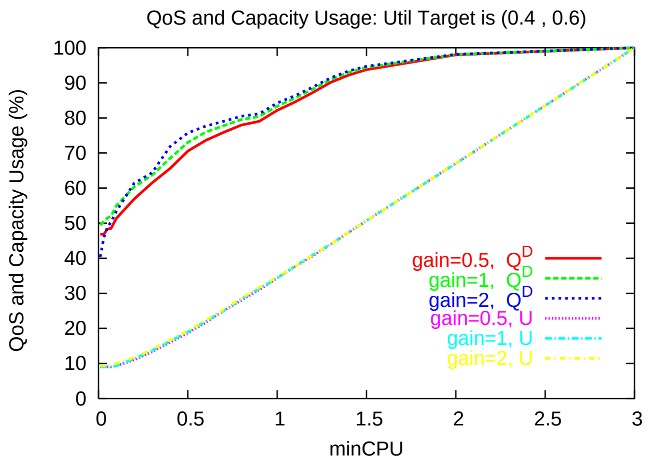

Fig. 10. Impact of Gain Parameter for Medium QoS Case

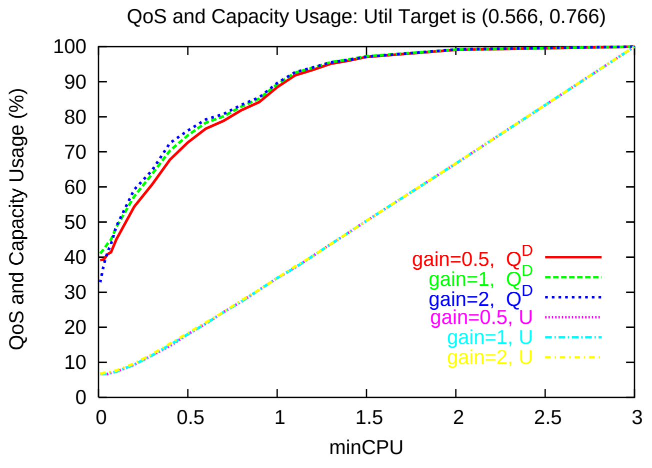

Fig. 11. Impact of Gain Parameter for Low QoS Case

CPU usage efficiency of underlying system capacity. This is somewhat counter-intuitive. We would expect the high QoS scenario to be much less efficient. From more detailed results we found that the workload spent significant portions of time with its *minCPU* allocation and not in its targetted utilization range. As the value of *minCPU* parameter decreases more time is spent in the targetted utilization range so we observe greater relative differences in U with the high QoS scenario being less efficient than the lower QoS scenario.

B. *minCPU and maxCPU Parameters versus* Q^D, Q^P *and* S^P *Service Metrics*

This section takes a closer look on the impact of *minCPU* and *maxCPU* parameters in workload manager controller algorithm on the service metrics Q^D (percentage of QoS satisfied demand), Q^P (percentage of time intervals with QoS satisfied demand), and S^P (percentage of time intervals with satisfied demand). Figures 12 through 14 illustrate these metrics for $maxCPU = 3$, 4, and 5 for the high, medium, and low QoS scenarios, respectively.

Consider Figure 12. The x-axis shows the *minCPU* value that is required to achieve a particular Q^D value on the y-axis. Several curves are shown. Three of the curves, that correspond to different values of $maxCPU = 3$, 4, and 5, show the relationship between the $minCPU$ parameter setting and achievable QoS satisfied demand metric Q^D. Two curves show the relationship between $minCPU$ parameter and Q^P and S^P metrics for values of $maxCPU = 3$, 4, and 5. Since the values for Q^P and S^P are nearly identical for each of the $maxCPU = 3$, 4, and 5 cases, respectively, only one curve is shown for all three considered values.

We now consider $Q^D = 80\%$ for more detailed discussion. Our choice is motivated as follows. As discussed in Section IV, a *minCPU = 0.6* corresponds to the 95-percentile of CPU demand for the workload. Figure 7 shows that for *minCPU = 0.6*, 20% of CPU demand is incurred in intervals with demand variation greater than 1.1. This set of demands is likely to cause reallocations by the workload manager controller for all three QoS scenarios thereby affecting Q^D, i.e., giving a limit of Q^D near 80%.

Our earlier figures, Figures 9 through 11, show the relationship between Q^D, its corresponding value for *minCPU*, and its corresponding value for capacity usage U for the high, medium, and low QoS scenarios, respectively. The capacity usage curves are the most bottom curves in the figures. The U values are practically identical for the different values *gain* parameter. Here, we consider a *gain=1*. From these figures (Figures 9 through 11), $Q^D = 80\%$ corresponds to U = 35%, 25%, and 20% for the high, medium and low QoS scenarios, respectively. As expected the low QoS scenario makes the most efficient use of resources. Also, as expected, from Figures 12 to 14, we see that the Q^P and S^P values diminish as we shift from a high to low QoS.

Finally, to achieve a $Q^D = 80\%$ for the high QoS scenario with $maxCPU = 3$, 4, or 5 the workload manager controller must be configured with a $minCPU$=1.1, 1.0, 0.9 CPUs,

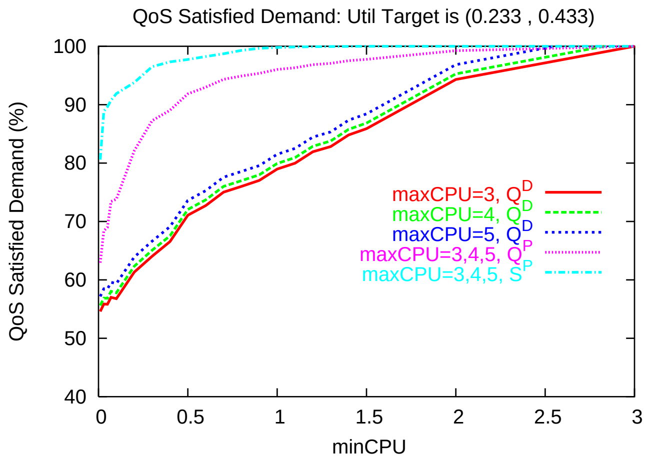

Fig. 12. minCPU v.s. Q^D, Q^P, and S^P for High QoS

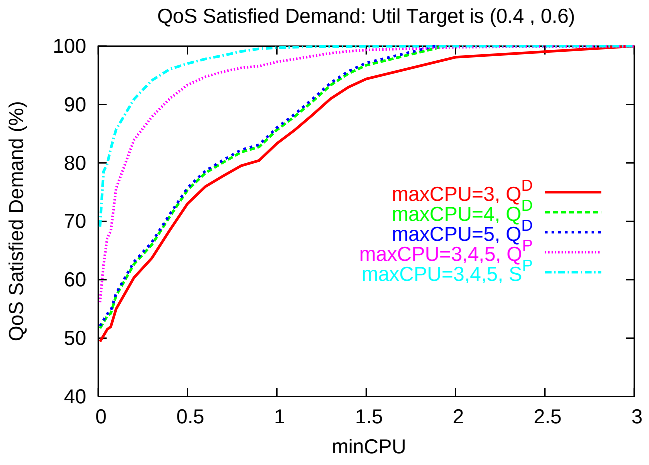

Fig. 13. minCPU v.s. Q^D, Q^P, and S^P for Medium QoS

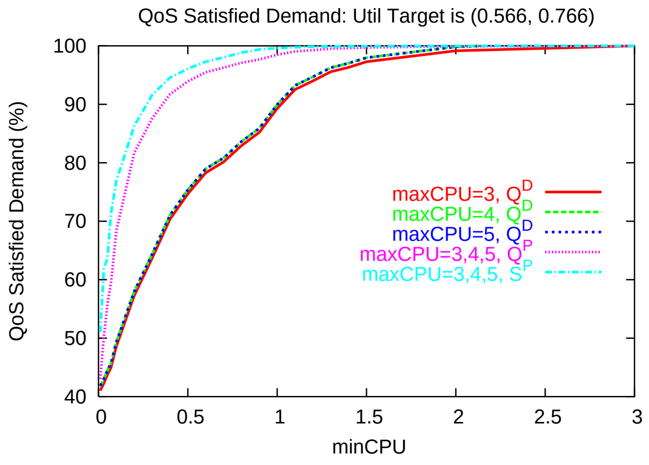

Fig. 14. minCPU v.s. Q^D, Q^P, and S^P for Low QoS

respectively. While these are lower CPU requirements than a fixed allocation of 3 CPUs, i.e., as in the reference case, we must still ask whether its possible to do better. The sum of $minCPU$ values over all resource containers must be less than the capacity of the resource pool. Large *minCPU* values may present an unintended limit on the number of workloads that can be associated with a shared resource pool whether or not the pool is heavily utilized.

The next section considers a modification to the workload manager control algorithm that provides similar QoS of satisfied demand and capacity usage while further reducing *minCPU* requirements.

VI. A New Fast Allocation Policy

We now consider a method to reduce the required *minCPU* parameter value while maintaining normalized QoS satisfied demand Q^D and capacity usage U. The method requires a modification to the workload manager control algorithm. We refer to the modification as a *fast allocation* policy. Basically, we aim to improve Q^D by increasing the allocation to *maxCPU* whenever the allocation is observed to be fully utilized. When the allocation is fully utilized then the true extent of demand is not known to the control algorithm; this is the most aggressive action possible to react to large changes in demand.

To implement the fast allocation policy, we modify policy 3 of the controller algorithm (see Section II) in the following way:

- If $upperAllocUtil \leq D_i/A_i$ then the controller attempts to increase the next step allocation:
  1) If $D_i < A_i$ then

$$A_{i+1} = A_i + gain \times (\frac{D_i}{upperAllocUtil} - A_i).$$

  If $maxCPU < A_{i+1}$ then $A_{i+1} = maxCPU$.

  2) If $A_i \leq D_i$ then $A_{i+1} = maxCPU$.

If the demand D_i is less than the assigned allocation (and hence the observed utilization of allocation is less than 100%), the fast allocation policy behaves in the same way as the original algorithm. Otherwise it increases the allocation to *maxCPU* as it is not known what the true demand is. With the original algorithm, it may take several intervals before the allocation is well matched with the true demand. With the fast allocation policy, the maximum entitled capacity is allocated in a single step. The new algorithm is therefore less efficient in its use of resources as it may take additional steps before the allocation is reduced to meet the workload's true demand. However it may provide a better quality of service for the workload.

Figures 15 through 17 show the relationship between *minCPU* and *maxCPU* with Q^D, Q^P and S^P for the high, medium, and low QoS scenarios, respectively.

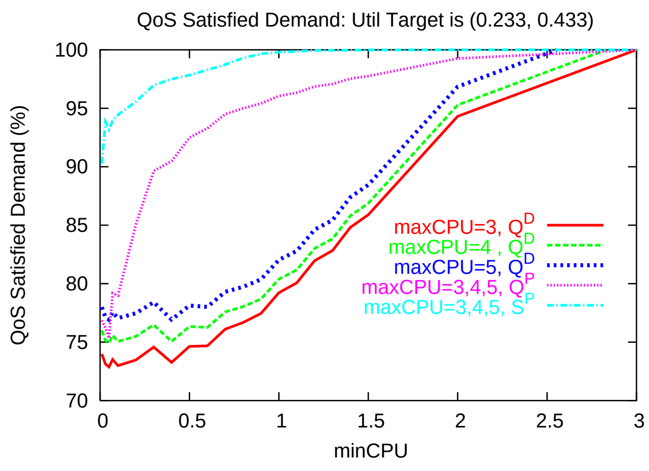

Fig. 15. *Fast Allocation Policy*: minCPU requirements v.s. service metrics Q^D, Q^P, and S^P for High QoS

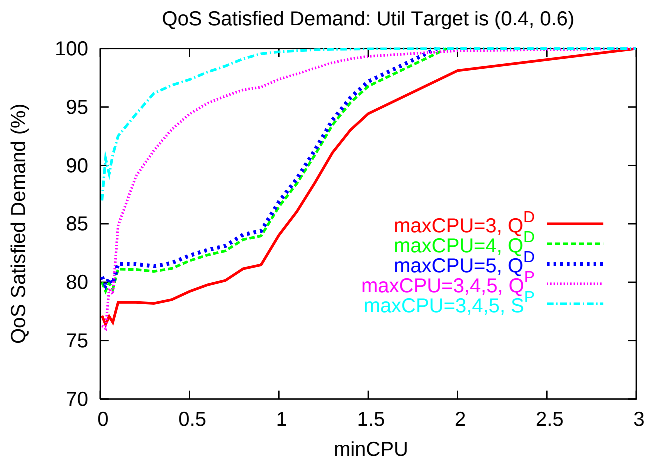

Fig. 16. *Fast Allocation Policy*: minCPU requirements v.s. service metrics Q^D, Q^P, and S^P for Medium QoS

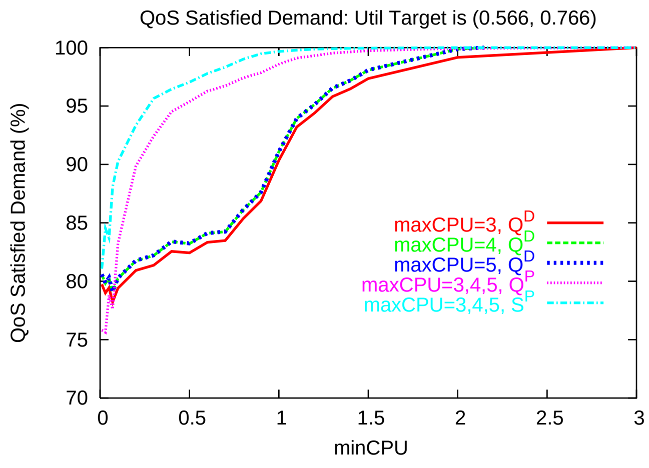

Fig. 17. *Fast Allocation Policy*: minCPU requirements v.s. service metrics Q^D, Q^P, and S^P for Low QoS

Figure 15 shows that even as *minCPU* approaches zero, i.e., *minCPU=0.01*, Q^D values remain near 75%. For the original workload manager algorithm the Q^D value approaches 55% for *minCPU=0.01*. Even greater improvements for Q^D are achieved for the medium and low QoS scenarios.

Though we may be tempted to exploit the low *minCPU* values when using the fast allocation policy, Figures 18 through 20 show that very low values of *minCPU* have an adverse impact on capacity usage U. The figures show the relationship between *minCPU* and capacity usage U for the high, medium, and low QoS scenarios, respectively. In each figure, as *minCPU* goes below approximately 0.2, the U value begins to increase, reflecting the decrease in capacity usage periodically caused by quickly changing the allocation to *maxCPU* value.

Now, we discuss Figures 15 through 17 in more detail.

- The high QoS scenario with a $Q^D = 80\%$ and a $maxCPU = 5$ requires a $minCPU = 0.85$ for the fast allocation policy as shown in Figure 15. The original algorithm could achieve a $Q^D = 80\%$ with a $maxCPU = 5$ and $minCPU = 0.9$ (see Figure 12). This is a 6% reduction in *minCPU*. Both approaches offer the same values for percentage of QoS satisfied time intervals Q^P and S^P that represent percentage of time intervals with QoS satisfied demands and satisfied demands respectively.
- The medium QoS scenario with a $Q^D = 80\%$ requires a $maxCPU = 4$ and a $minCPU = 0.2$ (to ensure a reasonable U) for the fast allocation policy as shown in Figure 16. The original algorithm requires a $maxCPU = 5$ and a $minCPU = 0.65$ (see Figure 16).

  Suppose that *minCPU* constrains the number of workloads that we can assign to a server. If the original algorithm requires a $minCPU = 0.65$ and the fast allocation policy requires a $minCPU = 0.2$ then we can potentially assign more than 3 times as many workloads to the server. A detailed analysis of workload patterns is necessary to predict how many workloads can actually be supported [1]. With the fast allocation policy, $minCPU$ can be less of a constraint.

  The Q^P and S^P values for the fast allocation policy are 90% and 94%, respectively. For the original algorithm with a $minCPU = 0.65$ the values are 95% and 98%. This is expected as fewer of the intervals with demands that are incurred when a workload transitions from idle to busy will be satisfied under the low *minCPU* value.

  However, we believe that Q^D is a more important metric as it gives the percentage of total demand where the utilization of allocation is less than *upperAllocUtil*. This identifies the portion of demands that receive the desired quality of service. Recalling from Section IV, for this workload, 1% of time intervals with higher CPU demands are be responsible for 10% of overall workload demand and 10% of time points with smaller CPU demands are be responsible for only 1% of the total workload demand, so metrics based on intervals alone can be misleading.

  The fast allocation policy could offer the same values for Q^P and S^P as original algorithm with an increased *minCPU* of 0.6 and a *maxCPU* of 4 or 5. Therefore, if Q^P and S^P metrics are matched for both the fast allocation policy and the original algorithm then the reduction in *minCPU* due to the fast allocation policy is 8% with Q^D increasing from 80% to 83%.
- The low QoS scenario behaves similarly to the medium QoS scenario. With a $Q^D = 80\%$ it requires a *max-CPU=4* and a $minCPU = 0.2$ (to ensure a reasonable U) for the fast allocation policy. The original algorithm requires $maxCPU = 5$ and $minCPU = 0.65$. Again the fast allocation policy reduces the $minCPU$ by more than a factor of 3.

  Capacity usage U for the fast allocation policy and original algorithm are 20% and 25%, respectively. Again, the fast allocation policy improves on efficiency and offers the same Q^D with a lower value for *maxCPU*.

  The Q^P and S^P values for the fast allocation policy under $minCPU = 0.2$ are 90% and 93%. For the original algorithm with a *minCPU* of 0.65 the values are 96% and 98%. The fast allocation policy can offer the same values for Q^P and S^P with increased $minCPU = 0.6$ and a $maxCPU = 4$ or 5. Therefore if Q^P and S^P metrics are matched the improvement is 8% and Q^D increases from 80% to 83%.

To summarize, we believe that the QoS satisfied demand metric is most representative of the QoS perceived by workloads. The fast allocation policy permits a 6% reduction in $minCPU$ for the high QoS scenario and a factor of more than 3 reduction in $minCPU$ for the medium and low QoS scenarios while maintaining the same Q^D value with respect to the original workload manager algorithm. Such reductions mean that there are greater opportunities for assigning more workloads to resource pools.

The original algorithm has higher values for Q^P and S^P, however we believe that with the fast allocation policy more demand that is carried forward is likely to be satisfied in the next interval which explains the correspondingly higher values for Q^D. With the original workload manager algorithm demands are more likely to be carried forward for many intervals and thereby incurring correspondingly lower values for Q^D.

VII. Related work

Recently there has been significant interest in resource management for shared resource environments. Issues include: capacity planning, which projects future capacity requirements for the pool; placement, which decides which workloads should share resources; allocation, which decides appropriate

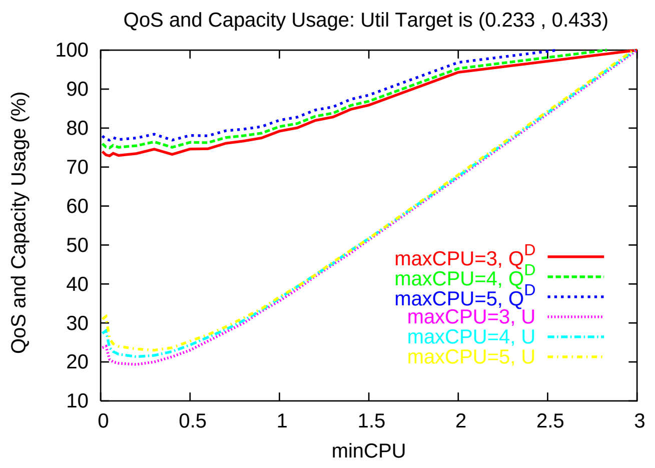

Fig. 18. *Fast Allocation Policy*: minCPU requirements v.s. capacity usage *U* for High QoS

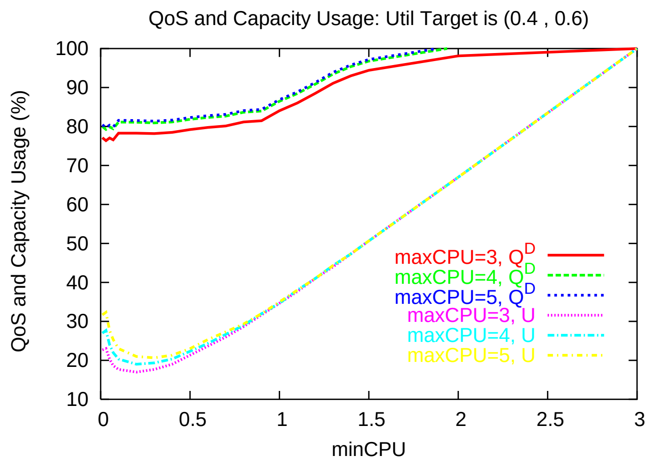

Fig. 19. *Fast Allocation Policy*: minCPU requirements v.s. capacity usage *U* for Medium QoS

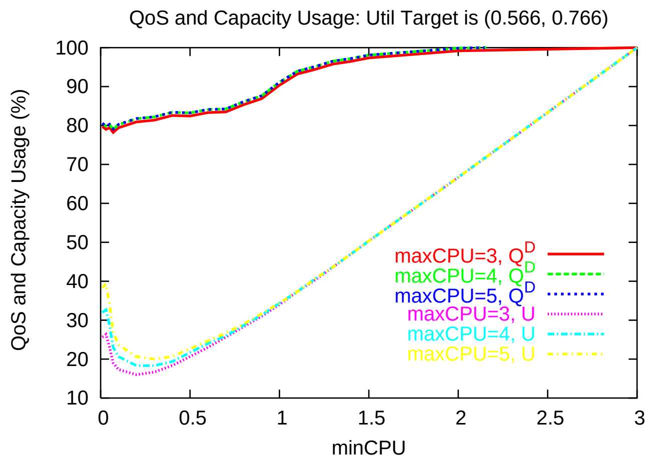

Fig. 20. *Fast Allocation Policy*: minCPU requirements v.s. capacity usage *U* for Low QoS

time varying resource allocations for application workloads; and arbitration, which decides how to partition capacity when demands exceed supply.

Our earlier work has focussed on capacity planning and workload placement [1] [2] [4]. We have assumed the presence of allocation and arbitration mechanisms of the kind that exist in industrial products [7] [8] [9] [10]. These implement control loops and are motivated by methods from control theory. Such products support demand based allocations, allocations based on workload response times or throughputs, priorities, entitlements, and other features. However the workload managers typically have control parameters that must be specified or configured in an off-line manner on a per-workload basis. More recently, adaptive controller technologies have been applied to allocation and arbitration [3] [5] [6] for such control loops. For example, Wang et. al. [5] use an adaptive controller to dynamically adjust the gain parameters of an integral controller based workload manager to better meet application level response time objectives. However values such as *minCPU* and *maxCPU* are still treated as fixed input parameters.

In this paper we explored the impact of *minCPU* and *maxCPU* on application resource access QoS and presented a fast allocation policy to improve access to capacity with lower values for *minCPU*. This provides the opportunity to support more workloads in the resource pool.

VIII. Conclusion

Resource pools are computing environments that offer virtualized access to shared resources. When used effectively they can align the use of capacity with business needs. In this paper we considered the impact of a workload controller on an application workload to be hosted in a shared resource pool. The workload controller algorithm has parameters including minimum and maximum allocation, gain, and a range of utilization of allocation values. We considered the impact of choices for such values in detail for a file server workload.

We found that it was necessary to select a relatively large value for the minimum to ensure a high application workload resource access quality of service. Unfortunately, selecting a large minimum allocation has the unintended side effect of limiting the number of workloads that can be hosted in the resource pool. We proposed a modification to the controller algorithm we named as a fast allocation policy. With the fast allocation policy the workload controller was able to provide high resource access quality of service with much lower minimum allocation values. For our example workload, in some cases, this provides the opportunity to increase the number of workloads supported by a resource pool by more than a factor of three.

The choice of maximum CPU allocation has an impact both on resource access quality of service and on capacity sizing. In general, we recommend choosing a maximum allocation that supports some high percentile of the application's demand. Our analysis of the file server workload showed that only a very small fraction of measurement intervals, e.g., much less than 1%, had demand values that were within a factor of two of the peak observed demand value. This is typical for enterprise workloads. Ultimately, the percentile that is chosen for a maximum allocation must depend on the importance of responsiveness for the workload during the infrequent bursts of high demand relative to the cost of providing the capacity.

The utilization of allocation range chosen for an application depends on the responsiveness requirements of a particular workload and the workload's characteristics. In general, the greater the potential number of concurrent users and the greater the inter-arrival time and service time variance the more bursty the workload. Greater burstiness requires lower utilization of allocation values to provide for specific average response times. The relationship between response times and ranges of utilization of allocation values is complex and must be verified empirically. Finally, for our file system workload trace we did not find gain to be an important parameter with respect to resource access QoS or the efficient use of resources.

Future work includes exploring the impact of the fast allocation policy on other workloads, e.g., application servers and databases. We will also investigate the impact of sudden changes in per-workload allocations on aggregate resource requirements for a resource pool.

ACKNOWLEDGMENT

We thank Tyler Easterling of Hewlett-Packard for providing us us with the file server traces used for this study. We also thank the anonymous reviewers for their helpful comments.

REFERENCES

[1] J. Rolia, L. Cherkasova, M. Arlitt, and A. Andrzejak *A Capacity Management Service For Resource Pools*, Proceedings of ACM WOSP 2005, Palma, Illes Balears, Spain, July 12-14, 2005, pages 229-237. Also available as Hewlett-Packard Labs Technical Report HPL-2005-1.

[2] A. Andrzejak, J. Rolia and M. Arlitt, Bounding Resource Savings of Utility Computing Models, HP Labs Technical Report HPL-2002-339, 2002.

[3] J.L. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury, Feedback Control of Computing Systems, Wiley-Interscience, 2004.

[4] J. Rolia, A. Andrzejak, and M. Arlitt, *Automating Enterprise Application Placement in Resource Utilities,* 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM 2003, Ed. M. Brunner and A. Keller, Springer Lecture Notes in Computer Science, Volume 2867, 2003, pp. 118-129.

[5] Z. Wang, X. Zhu, and S. Singhal, *Utilization vs. SLO-Based Control for Dynamic Sizing of Resource Partitions*, to appear in the 16th IFIP/IEEE Distributed Systems: Operations and Management(DSOM 2005), October, 2005.

[6] Magnus Karlsson, Xiaoyun Zhu, and Christos Karamanolis, *An adaptive optimal controller for non-intrusive differentiated services in computing services,* in Proc. of the 5th International Conference on Control & Automation (ICCA 2005), June, 2005.

[7] HP Process Resource Manager, http://h30081.www3.hp.com/products/prm/index.html

[8] HP-UX Workload Manager, http://h30081.www3.hp.com/products/wlm/index.html

[9] IBM Application Workload Manager, http://www.ibm.com/servers/eserver/xseries/systems_management/director_4/awm.html

[10] IBM Enterprise Workload Manager, http://www.ibm.com/developerworks/ autonomic/ewlm/