Proactive Management of Service Instance Pools for Meeting Service Level Agreements

Kavitha Ranganathan and Asit Dan

IBM T J Watson Research Center,
19 Skyline Drive, Hawthorne, NY, 10025, USA
{kavithar, asit}@us.ibm.com

Abstract. Existing Grid schedulers focus on allocating resources to jobs as per the resource requirements expressed by end-users. This demands detailed knowledge of application behavior for different resource configurations on the part of end-users. Additionally, this model incurs significant delay in terms of the provisioning overhead for each request. In contrast, for interactive workloads, services are commonly pre-configured by an application server according to long-term steady-state requirements. In this paper, we propose a framework for bridging the gap between these two extremes. We target application services beyond simple interactive workloads, such as a parallel numeric application. In our approach, end users are shielded from lower-level resource configuration details and deal only with service metrics like average response time, expressed as SLAs. These SLAs are then translated into concrete resource allocation decisions. Since demand for a service fluctuates over time, static pre-configurations may not maximize utility of the common pool of resources. Our approach involves dynamic re-provisioning to achieve maximum utility, while accounting for overheads incurred during re-provisioning. We find that it is not always beneficial to re-provision resources according to perceived benefits and propose a model for calculating the optimal amount of re-provisioning for a particular scenario.

1 Introduction

Existing Grid [1,3,4] scheduling technologies – as described in commercial products [5,12], prototypes [15,7,8], requirement specifications [13,14] and papers [11, 16,17, 9] – have primarily focused on allocation of resources to incoming jobs as per the resource requirements expressed directly by end users. An experienced end-user of an application, say a scientist submitting a numerically intensive application, is quite knowledgeable of the application execution behavior over different resource configurations. Thus the scientist is able to manually translate the requirements on timeliness and data size into a set of requested resources over which the application is to be run. Current resource provider capabilities do not permit a client to use high level objectives such as a deadline or desired throughput. Note that a service could be an invocation of an interactive workload or a longer running numeric intensive application. As pointed out in [2], to bridge the gap between a high level client request and the resource provider capabilities to make available any requested resources, an intermediate layer

B. Benatallah, F. Casati, and P. Traverso (Eds.): ICSOC 2005, LNCS 3826, pp. 296 – 309, 2005.

must translate high level objectives to detailed resource requirements based on application execution profiles.

To illustrate a long running parallel application service, consider a financial application for portfolio risk evaluation consisting of two phases as described in [2]. The first phase invokes a service that solves a large system of linear equations, and is implemented as a parallel, tightly-coupled, compute and network intensive computation. The result of this phase along with new input is used to compute the risk profiles of a set of trades in the second phase. The computation of this phase is organized as a master-worker interaction: each worker node independently computes the risk profile for a set of trades, and the master assembles the information received from workers into the final result. The overall computation must be completed by a fixed deadline (say 7:00 AM of each trading day). The client application breaks down the overall objective, by setting service level goals (such as completion time or number of trade risk computation per second) for each phase of the portfolio computation.

Setting up a service instance with the required resource configuration involves acquiring the required set of resources, starting up a (parallel) application on these nodes, and finally after execution, shutting down the application and releasing allocated nodes. To avoid delay and overheads associated with provisioning resources on a per-request basis, an application service instance can be reused for serving another user request with similar requirements; as is done when managing interactive workloads. Note that an available pre-configured application instance may not always match the resource requirements of a new user request. Hence, the new user request may have to wait until a matching application instance is available. If no such instance has been pre-configured or existing instances are insufficient, a new instance could be created on-demand. The resources for the new instance may be obtained by destroying one or more less frequently used pre-configured instances. Once an application instance is acquired by an end-user, it is either used to run a single (long running) request, or is used to invoke a series of short operations by the end-user. The end-user therefore, expresses an SLA that not only translates to resource configuration requirements of an instance, but also defines service level objectives like the wait time to acquire a matching service instance.

In this paper, we propose a layered framework for managing application level SLA objectives for all types of services (not just interactive workloads), including invocation of parallel applications. The framework addresses the translation of end-user SLA requirements into concrete resource requirements to be used for configuring a service instance based on a prior application execution profile. It also addresses (1) scheduling user requests to matching available pre-configured service instances, (2) pro-active management of a pool of pre-configured service instances for meeting SLA objectives on waiting time, (3) allocation and de-allocation of physical resources to service instances, and (4) provisioning and de-provisioning of service instances.

We then explore key issues and strategies in pro-active management of service instance pools, taking into account SLA objectives. We incorporate the provisioning overhead - often ignored in steady state analysis – as well as gauge the effect of different user request arrival patterns. We find that it is not always beneficial to re-provision resources according to perceived benefits. In fact, in certain cases the re-provisioning overheads make it detrimental to adapt service pools according to user request arrival patterns. We hence propose a model for calculating the optimal amount

of re-provisioning for a particular scenario. Preliminary simulation results also show that we are able to effectively manage service instance pools, based on user defined SLAs. Additional experimental results are described in [18].

The rest of this paper is organized as follows. Section 2 proposes a layered framework for managing application level end-user SLA and scheduling user service requests. Section 3 provides details on dynamic management of service pools, including SLAs and workloads used, algorithm tested, and a model for calculating the optimal level of re-provisioning. Section 4 contains the experimental setup, parameters and simulation results. We conclude in Section 5.

2 Overview of the Architecture for Service Instance Scheduling and Management

We now detail the proposed layered architecture for managing application level SLAs using pre-configured service instances (See Figure 1). Before invoking a service, a client application establishes a SLA with the service provider, that expresses not only the capabilities of the service instance to be assigned, but also timeliness in receiving a service instance. The SLA for service capabilities can be expressed either as a deadline or as the number of transactions per unit time supported by such an instance.

The primary components of this architecture and their interactions are detailed below.

Application Service Level Manager (ASLM): A service client interacts with the ASLM for establishing a new SLA, and subsequently monitors the status of the SLA. Our prototype supports the WS-Agreement protocol [10] for the above interactions, which additionally includes customizing predefined agreement templates to create a new SLA.

In addition to establishing the SLA, the ASLM translates higher level SLA objectives into detailed resource requirements to be used for configuring a service instance in support of this SLA. To estimate required resource configurations, the ASLM maintains an application execution profile consisting of multiple observed performance points for various resource configurations. The focus of this paper is not on this translation method, and hence, details of how to obtain the execution profile, bounding the number of observations to be obtained for defining execution profile, etc. are not discussed here. We mention in passing that the execution profile captures only the effects of changes in key resource attributes, such as the number of nodes, processor MIPS, and memory size per node.

Service Instance Request Scheduler (SIRC): Once an SLA is established, a service client requests a service instance that is configured according to the SLA. Depending on the type of a service, this request may be explicit or implicit. If the service instance is to be used for a single invocation (e.g., a long running application), the request for a service instance can be combined with the service invocation request, i.e., assignment of service instance is implicit. Alternatively, a client may request a binding to an explicit service instance, and perform multiple service invocation on this instance (as in the second phase of the financial application example, discussed in Section 1).

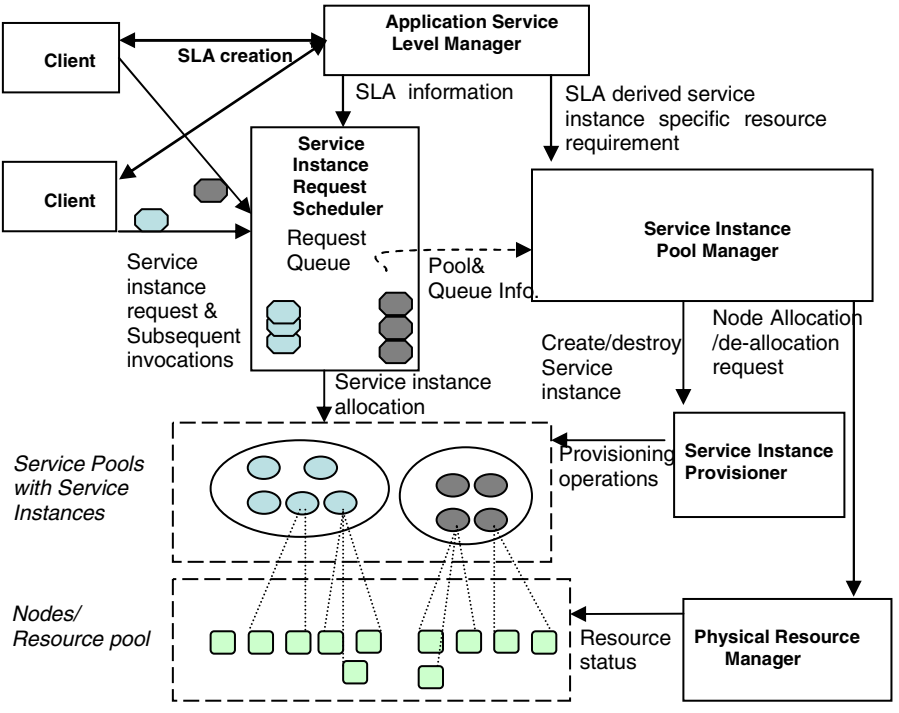

Fig. 1. Layered architecture for service Instance scheduling and management

The SIRC assigns a matching service instance to an incoming service instance request, if such an instance is available. Otherwise, it queues the incoming request, and prioritizes requests in the queue to meet SLA waiting time objectives. The service instance pool manager monitors the queuing delay, and possible SLA violations, and pro-actively signals provisioning of new service instances.

Service Instance Pool Manager (SIPM): The SIPM makes dynamic decisions on the number of services instances to be maintained for each service type. Multiple SLAs may specify the same set of service objectives, referred to as a service type. The decision is based on the business values associated with SLA objectives, required resource configuration for each service instance derived by ASLM and the current state of objectives. Following this decision, the SIPM creates and/or destroys service instances of different service types by invoking the two components discussed next, the Physical Resource Manager and the Service Instance Provisioner. The focus of the current paper is on the strategy of managing service instance pool, and we will provide more details in Sections 3 and 4.

Physical Resource Manager (PRM): The PRM manages allocation of nodes to service instances, and is invoked by the SIPM to allocate nodes for a new service instance to be created or to release nodes when a service instance is destroyed. Upon destruction of a service instance, the nodes are made available for reallocation.

Service Instance Provisioner (SIP): Actions involved in provisioning a service instance depends on the type of the service, as well as differences across services sharing the same resource pool. For a relatively simple scenario, this involves merely starting up an application on one or more nodes. For a parallel application this may involve synchronization with the master node. In a more complex scenario, where different applications require different execution environments (e.g., J2EE version), a completely new software stack needs to be loaded for reassigning a node to another application. In all scenarios, once a service instance is created, the SIRC is notified of this new instance.

3 Service Instance Pool Management

We now discuss issues involved with the dynamic management of service pools. The characteristics of the service request streams (henceforth referred to as the workload) and SLAs used play a key role in the implementation and effectiveness of dynamic service instance pool management. We then propose an algorithm that is used by the SIPM to manage these service instance pools. Other key issues explored are the overheads involved in re-provisioning service instances i.e. moving nodes across service pools – and the extent of adaptability for the SIPM.

3.1 Request Workloads, SLAs and Key Issues

Workload Characteristics: If requests for each particular service-type arrive at a steady rate (say according to a uniform or Poisson distribution) then it is relatively easier to decide the number of service instances to be instantiated per service, to meet the SLA goals. Many systems today assume steady-state parameters to calculate resource allocation. However, in many cases, the workload may change over time, either gradually or suddenly (for example, there might be a sudden spike in demand for one particular service) warranting the increase in the number of instances of certain service types. A successful adaptation technique should be able to detect the changes in the workload and re-provision instances accordingly.

We look at both cases in our experiments: (1) a steady-state scenario where requests for different services may arrive at different rates but follow Poisson distributions and (2) when the workload varies over time.

Higher Level SLA Objectives: The Service Level Agreements between the provider and client could take a number of forms [10]. In this paper we assume business values associated with objectives are defined as explicit utility functions, where a client specifies how much utility is gained (or in other words, how much she is willing to pay) for a certain response time for acquiring the service instance. The same utility function also applies to the scenario of how much penalty should be assessed for deviations from the specified goal.

Key Issues in Dynamic Adaptation: Often, there might be inherent costs associated with re-provisioning a node. At one extreme, the overheads might be negligible if switching from one service to another just involves linking different libraries. On the other extreme, it may involve, draining the node of current jobs and software, I/O op-

erations to load the software for the new service and extensive re-configuration. Thus the "dead time" defined as the time when the node is unavailable for any service, could be substantial in many cases of re-provisioning. Moreover, re-provisioning might require I/O or other operations which access bottleneck resources like the network or shared disk access. Thus, the time to re-provision '*k*' nodes may not be the same as the time to re-provision one node. Depending on how costly the re-provisioning step is, and the relative increase in utility it may actually be disadvantageous to re-provision more than a certain number of nodes at a time. Our proposed model for estimating these costs is described later.

3.2 Algorithm for Incremental Adaptation

We now describe the algorithm employed by the SIPM to make decisions regarding the number of service instances to maintain for each service pool. Since the SIPM has access to the different utility functions for each service type, it can make re-provisioning decisions with the aim of maximizing utility (henceforth called revenue) across all resources it controls. We assume that we know how long a particular service instance will be used by a client (This can be derived either by employing predictive techniques or by requiring that the user submit an estimate of the usage time).

We first establish some terms that will be used in the algorithm. As explained earlier, the revenue acquired by a service is a function of the response time (as defined in the SLA). Hence the revenue gained for time interval t0-t1 is captured in the following expression:

*Revenue(t0-t1) = Number of requests fulfilled in t0-t1 * utility_function(average response time for fulfilling requests in t0-t1)*

Thus, given a request queue for a particular service type, the number of service instances (i.e. the size of the service instance pool) and the estimated run-time for each request in the queue; the predicted average response-time to obtain a service instance, can easily be calculated. This can then be used to derive the predicted revenue.

Predicted Revenue = function of (Size of Service Instance Pool, Run time estimates of Requests in Queue).

Thus, if a Service Instance was added to a pool that already had *s* Service Instances, the advantage of adding that new Instance (the 'Incremental Revenue' gained) could be calculated as follows :

Incremental Revenue = (Predicted Revenue with s+1 instances) – (Predicted Revenue with s instances).

The algorithm for incremental adaptation is then a simple calculation to maximize revenue, as the following pseudo code describes :

At each time period (the periodicity factor is discussed in the next section):

1) Each service instance pool donates at-least *'k'* nodes to a common virtual pool. If a certain service instance pool does not contain enough resources to contribute '*k*' nodes to the common pool, it does not contribute nodes in that iteration, but is however considered a candidate to receive nodes. Note that nodes are not de-provisioned at this stage (The size of *'k'* is discussed shortly).

2) Now the SIPM calculates how best the nodes in the common pool could be re-distributed across service instance pools. Assume here that each service instance requires one node. Variations are discussed below.

For each node (n) in the common pool :
For each Service Instance Pool (p):
Calculate 'Incremental Revenue' gained if node n is added to Service Instance Pool p
Assign node n to Service Instance Pool with maximum 'Incremental Revenue'
Update size of that Service Instance Pool
Record new assignments

Note that if each service type requires more nodes than one, then instead of one node, groups of nodes of the required size, are considered at each iteration.

3) Re-provision nodes according to the new assignment that was calculated. (Note that actual re-provisioning only happens in Step 3).

The incremental adaptation algorithm, reassigns nodes to where they might be most useful, but at the same time, ensures that this re-assignment is gradual. Temporary variations in the request workload may cause a small number of service instances to be re-provisioned. Massive re-structuring can only occur if there is a sustained change in external parameters like SLAs or service demand.

3.3 Model for Estimating Optimal Amount of Simultaneous Re-provisioning

A key factor for dynamic adaptation is that while a node is being re-provisioned, it is unavailable for providing any service. These so called "dead times" could be significant. Moreover, as discussed earlier, concurrently re-provisioning multiple nodes could lead to larger dead-times than when only one node is re-provisioned. Hence this cost has to be factored in when deciding the adaptation algorithm parameter of how many nodes to consider for re-provisioning in each iteration of the algorithm. The following model aims to estimate this parameter, by calculating the loss in revenue caused by the dead times and the gain in revenue, resulting from running a different service on the nodes.

Assume there are two service types, *s1* and *s2*, which currently generate revenue *r1_old* and *r2_old* per time interval *t*. There are *n* service nodes in the system and they are initially partitioned into the two service types (*x1* nodes provide service *s1*, and *x2* provide service *s2*). Suppose we now want to re-provision *k* nodes from serving *s1* to serving *s2* and the new revenues that will be generated by each are *r1_new* and *r2_new* per time interval. (To recall, the revenue generated is a function of the average response time, which will change when the number of instances per service type changes).

Now the cost of moving *k* nodes from service type *s1* to service type *s2*, is at the very least the revenue lost by not serving *s1*, for the duration of the dead time of those *k* nodes. Assume that if it takes *T* seconds to re-provision one node, it will take *T + delta * (k-1)* to re-provision *k* nodes, where *delta* is the simultaneous re-provisioning cost factor. Hence the dead time for one node = *T* where as the dead time for *k* nodes is *kT + k(k-1)*delta* . Thus the revenue lost if *k* nodes are re-provisioned = *(kT + k(k-1) * delta) * r1_old*.

The benefit gained by the re-provisioning can be quantified as the increase in revenue, that is Revenue(old) – Revenue(new). We assume here, that there is enough demand to use up all nodes. Suppose, the time-interval for periodically adapting is *Ta*. The revenue made by the nodes for the non-adaptive case: Revenue(old) = *(r1_old * s1 + r2_old * s2) * Ta*. The Revenue made in the adaptive case: Revenue(new) can be split into 2 stages T1 and T2 where Ta = T1 + T2. T1 is the time for the re-provisioning to occur [which was earlier calculated as = *T + delta * (k-1)*] and T2 is the time when the re-provisioning has already succeeded.

During T1, there are *x1-k* nodes serving *s1*, *x2* nodes serving *s2* and *k* nodes unavailable.

*Therefore, Revenue(T1) = ((x1-k) * r1_new + x2 * r2_old) * T1*
*Similarly, Revenue (T2) = ((x1-k) * r1_new + (x2 + k) * r2_new) * T2*

Hence, the increase in revenue for a particular *k* can be calculated as *Revenue(T1) + Revenue(T2) – Revenue(old)* . The optimal value for *k* can be derived by calculating the maxima of this resulting function. Figure 3 plots the revenue gain for some sample values of *x1*, *x2*, *utility curves* and *delta*. It is clear from the figure that for a given *delta*, there is a certain range for *k*, when it is most beneficial to re-provision those many nodes simultaneously. Re-provisioning more nodes than this value leads to a steady decline in revenue and even to losses.

Frequency of Adaptation: Closely related to the overheads of re-provisioning is the frequency with which adaptation occurs. In our architecture the SIPM periodically re-calculates if nodes need to be re-provisioned. If this frequency is too high, then minor fluctuations from the steady state may cause unnecessary re-provisioning; if too low, then the adaptive machinery may be too slow to react to genuine surges.

We tackle this issue by using an incremental adaptation process. At each time-period (which is relatively short) only a certain number of nodes are considered for re-provisioning. If the surge that triggered the re-provisioning was small or short-lived, this adaptation is corrected in the next time interval. If however the surge is a genuine one, and has lasted beyond a couple of time-periods, subsequent adaptations further create the necessary services.

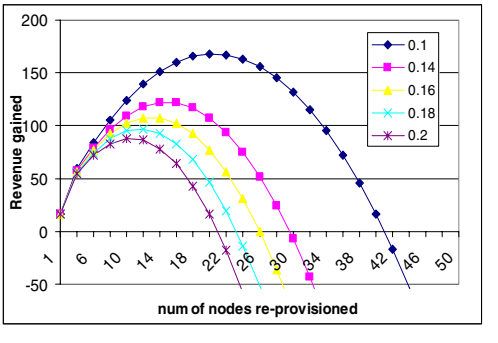

Fig. 2. Revenue gained for different values of Delta. Total nodes = 100, r1 old and r1 new = 2, r2 old and r2 new = 4, T =1 and Ta = 10.

This incremental adaptation has a twofold advantage: (1) temporary aberrations to the steady-state workload don't falsely trigger a huge amount of costly re-provisioning (2) by re-provisioning smaller batches of nodes at a time (around the optimal value of k as explained earlier), we decrease the dead times of nodes and hence maintain higher revenues.

4 Experiments and Results

To test the effectiveness of our dynamic adaptation algorithm, we simulate a cluster of nodes providing different services. Our in-house simulation program generates various service requests, and allocates and executes them on the services running on the cluster. We then use the prototype we have built for incremental-adaptation; the SIPM, to manage the service instances provided on this simulated cluster. We measure both the net revenue gained by adaptation and improvement in performance in terms of response time.

4.1 Experimental Setup

In the start of each experiment, all the nodes in the cluster are equally pre-divided into two logical service instance pools. There is a fixed utility function associated with each service type, and we assume here that both service types run on the same number of nodes. Client requests for a service are generated according to a Poisson distribution. Each service instance pool has an associated queue of requests waiting to acquire a service instance from that pool, and incoming requests are added to this queue as the simulation progresses. Requests are allocated to service instances by the Service Instance Request Scheduler, using a FIFO (First In First Out) algorithm.

The Service Instance Pool Manager (SIPM) kicks in periodically to re-provision service instances if needed. The input to the SIPM consists of the size of each service instance pool, the size of request queues for each service instance pool and the respective utility functions. The SIPM then calculates the desired size of each service instance pool, so as to maximize revenue. The Service Instance Provisioner then re-provisions nodes, so as to meet the new assignment. Note that nodes do not physically belong to certain pools, but form logical pools on the basis of the type of service instance they belong to.

The experimental parameters used are provided in Table 1.

Table 1. Parameters used in simulations

Experimental Parameter	*Value*
Total number of nodes in cluster	100 - 200
Number of requests per service type	varies by experiment ; 250-1000
Job run time	60 seconds
Nodes per service instance	5
Periodical adaptation	Varies; typically 200 seconds
Time to re-provision one node	Varies; 5 – 100 seconds
Delta : factor that determines overhead of re-provisioning n nodes simultaneously	Varies; 0.5 - 20

4.2 Simulation Results

We first want to compare our incremental-adaptation approach against the case were no adaptation takes place, to ascertain whether dynamic adaptation does result in higher revenues for the provider. We then go on to study parameters like workload variations and cost of re-provisioning, that might impact the algorithm. The results are an average of three runs, where the varying factor in each run was the generated workload. We did not find significant variations between runs.

Effectiveness of Incremental Adaptation
We first consider the case of two services types, S1 and S2, each offering 10 service instances each. We test two cases (1) where both services have the same utility function but different request arrival rates and (2) where they have the same request arrival rate but different utilities.

(A) Heterogeneous Request Arrival Rates: We specify the same utility function for both services but different request arrival rates. Maximum revenue is gained if requests for a service instance do not incur any delay. If the response time goes beyond 40 seconds, then there is a penalty associated with that request. The request arrival rates for the two services differ by a factor of two. Requests for S1 arrive twice as often as requests for S1 (Poisson inter-arrival time for S1 = 5, S2= 10).

Table 2. Performance comparison of static provisioning and SIPM's dynamic re-provisioning for heterogeneous request arrival rates

	Avg. Response Time (secs)		Revenue ($)	
	Static	SIPM	Static	SIMP
S1	578	31	-2662	44
S2	02	14	95	63
Total	na	na	-2567	107

Table 2 contains the average response time (the time it takes to allot a particular service instance to a request) and net revenue gained when the above utility function is used, for both the static case when no adaptation takes place, and the dynamic case where the SIPM re-provisions service instances to increase revenue gained. As seen, in the static case, the average response time for S1 in very high, leading to a large penalty imposed on the provider. When the SIPM is used, the response times for S1 are effectively brought down to 91 seconds whereas the response time for S2 goes up slightly, resulting in positive revenues. Note that the revenue numbers are entirely dependant on the utility function used to interpret the gains, but the average response times are independent of whatever utility function is used.

These results can be explained as follows: since there are many more outstanding requests for S1 than S2, the SIPM successfully detects this, re-provisions nodes serving S2 to serve S1 and decreases wait-times for S1 requests.

(B) Heterogeneous Utility Functions: For this experiment, requests for both services arrive at the same rate, but the services have different utility functions as shown

in Figure 3(left). S2 is a relatively more critical application than S1 and hence the client offers higher revenues if the ideal response time is met for S2. Thereafter, the utility curve for S2 decreases more rapidly than for S1. If the response time for S2 is greater than 100 seconds, the provider incurs a penalty. We model the same request arrival rate for both services: a Poisson arrival rate with inter-arrival time of 6 seconds.

Table 3 provides the results of this experiment for both the static and SIPM cases. As the results show, the SIPM is able to effectively increase net revenue earned, as compared to the static node distribution case. Figure 3(right) shows the number of service instances as the simulation progresses, for one particular run. Initially, the SIMP re-provisions some nodes to serve S2, since the revenue gained from S2 is higher. As the simulation progresses, the request queue for S1 grower longer (as lesser nodes now serve S1), effectively making it more lucrative to switch some nodes back to S1. The SIPM also detects small bursts in traffic and adapts slightly, bringing down the average response time considerably.

Effect of Workload Variations

To test how the SIPM reacts to a sudden increase in requests for one service type, we generated a workload where after a short while, requests for S2 start arriving more frequently. Requests for S1 have a constant inter-arrival time of 6 throughout the workload, where as after 500 seconds, S2 requests start arriving much faster (with an inter-arrival time of 2 seconds). S2 jobs cease arriving at time 1150.

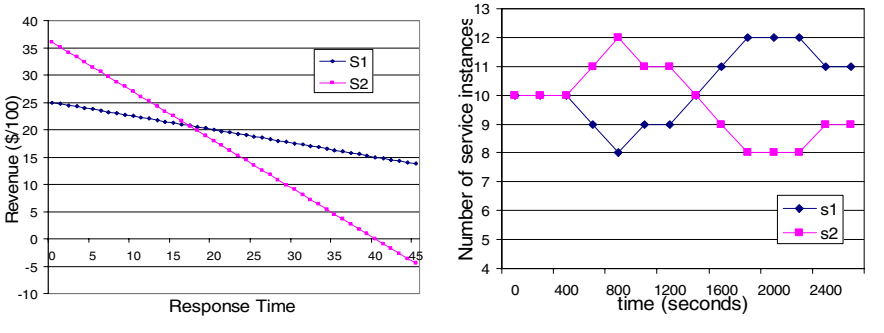

Fig. 3. (left) Utility functions used for re-provisioning and (right) number of service instances as simulation progresses

Table 3. Performance comparison of static provisioning and SIPM's dynamic re-provisioning for heterogeneous utility functions

	Avg. Response Time (secs)		Revenue ($)	
	Static	SIPM	Static	SIMP
S1	178	66	-98	41
S2	165	39	-56	18
Total	na	na	-154	59

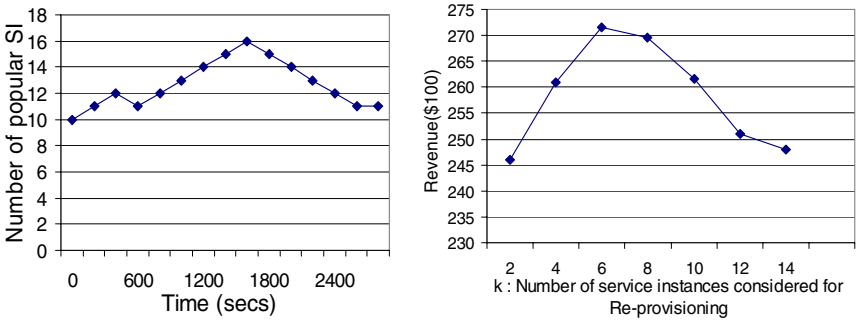

Fig. 4. (left) Number of service instances as simulation progresses for an unstable workload (right) Revenue generated as the amount of simultaneous re-provisioning increases

Figure 4(left) shows the number of service instances for S2 as the simulation progresses (the remainder of a total of 20 instances are S1 instances). As can be seen, the SIPM is quickly able to detect the surge in S2 requests (at time 600) and increases service instances of S2. However, once most S2 requests are met, nodes are switched to S1 instances (starting at time 1600), to better respond to S1 requests.

Effect of Cost of Re-provisioning

While we have ascertained that the SIPM is able to re-provision nodes according to workload fluctuations and utility definitions, we want to study the effect of the overheads of re-provisioning.

The next experiment quantifies the effect of simultaneously re-provisioning instances, the *k* factor, as explained in Section 3.3. Each service type in this experiment starts off with 20 instances each, and *k* instances (that is k/2 from each service instance pool) are periodically considered by the SIPM for re-provisioning. One service type is defined as having consistently higher returns than another, prompting the SIMP to re-provision as many nodes as permitted by the value of *k*. Figure 4(right) plots the revenue gained as the number of instances being simultaneously re-provisioned (*k*) is increased.

As can be seen from the figure, there is a distinct advantage in increasing *k* to a certain point. The adaptation process succeeds in generating higher revenues. But beyond the threshold value ($k = 6$ in this case) it is less advantageous to re-provision more nodes simultaneously. This is because, as explained in our model in Section 3.3, the loss in revenue caused by the dead-times over-weigh whatever increase in revenue the new service type generates. It should be noted here that the value of delta determines the cost of simultaneous re-provisioning. In future work, we plan to run experiments on a real test-bed to obtain realistic ranges for delta.

5 Conclusions and Future Work

Existing scheduling solutions require end-users to express exact resource requirements for each request in the form of a job submission. We have proposed a framework where end-users need not be aware of specific resource configurations needed to

realize their service level objectives. Our framework translates high-level end-user service level objectives like desired response-time to specific resource scheduling and provisioning actions based on application execution profiles. Since configuring a new service instance, especially for parallel applications, can incur both delay and overhead, the proposed framework furthermore, reuses existing pre-configured service instances in serving a new user. An end-user first establishes a SLA, and receives a service instance configured to meet SLA objectives, which is then used for subsequent invocations. To avoid a long delay in acquiring a service instance, SLAs also specify response time objectives in acquiring a new instance.

Additionally, our framework also enables dynamically re-provisioning nodes to meet SLAs provided by users. To this end, we put forth an incremental adaptation algorithm for dynamically re-provisioning services and an analytical model for estimating the optimal amount of re-provisioning. Initial simulation results to evaluate our prototype show that not only is it successful in adapting service instance pools, for maximizing utility, but also that the optimal amount of adaptation depends on the cost of provisioning.

In future work, we plan to run more experiments on test-beds using real workloads to better quantify the overheads of simultaneously re-provisioning nodes.

Acknowledgements. The authors would like to acknowledge the contributions of and thank Cait Crawford, Liana Fong, Kevin Gildea, Alan King, H. Shaikh, and Annette Rossi on the broader formulation of reusable parallel application service instances.

References

1. I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," Globus Project 2002.
2. A. Dan, C. Dumitrescu, and M. Ripeanu, “Connecting Client Objectives with Resource Capabilities: An Essential Component for Grid Service Management Infrastructures”, 2nd International Conference on Service Oriented Computing (ICSOC), November 2004, New York, NY.
3. A. S. Grimshaw and W. A. Wulf, "The Legion Vision of a Worldwide Virtual Computer," Communications of the ACM, vol. 40, pp. 39-45, 1997.
4. I. Foster, "The Grid: A New Infrastructure for 21st Century Science," Physics Today, vol. 55, pp. 42-47, 2002.
5. I.B.M Corporation. IBM LoadLeveler: User’s guide. Technical report, IBM, September 1993.
6. H. Ludwig, A. Keller, A. Dan, and R. King, "A Service Level Agreement Language for Dynamic Electronic Services," presented at 4th IEEE International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems (WECWIS'02), Newport Beach, California, USA, 2002.
7. R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed Resource Management for High Throughput Computing", Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing, July 28-31, 1998, Chicago, IL
8. The Globus Resource Allocation and Management http://www.unix.globus.org/toolkit/docs/3.2/gram/ws
9. C Liu, L Yang, I Foster, D Angulo, “Design and Evaluation of a Resource Selection Framework for Grid Applications”, HPDC, Edinburgh, July 2002.

10. A. Andrieux, C. Czajkowski, A. Dan, K. Keahey, H.Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, M. Xu., Web Services Agreement Specification (WS-Agreement). Version 1.1, Draft 20, June 6th 2004.
11. D.Feitelson, L Rudolph and U Schwiegelshohn, Proceedings of the Job Scheduling Strategies for Parallel Processing, 10th International Workshop, JSSPP 2004, New York, NY, USA, June 13, 2004.
12. C Smith, "Open Source Metascheduling for Virtual Organizations with the Community Scheduler Framework (CSF) , White Paper, Platform Computing Inc.
13. The JSDL Specification https://forge.gridforum.org/projects/jsdl-wg/document/draft-ggf-jsdl-spec/en/21
14. The Globus Resource Specification Language RSL v1.0: http://www-fp.globus.org/gram/rsl%5Fspec1.html
15. The NorduGrid Project. http://www.nordugrid.org
16. J. Gehring and A. Reinefeld. MARS - a framework for minimizing the job execution time computing environment. Technical report, Paderborn Center for Parallel Computing, Jan 1995.
17. G. Allen, D. Angulo, I. Foster, G. Lanfermann, and C. Liu. "The Cactus Worm: Experiments with dynamic resource discovery and allocation in a Grid environment", International Journal of High Performance Computing Applications, 15(4),Jan 2001.
18. K. Ranganathan and A. Dan, "Proactive management of Service Instance Pools for meeting Service Level Agreements", Technical Report, I.B.M - RC23723, September 2005.