Adaptive Resource Scheduling for Network Services

Byoung-Dai Lee and Jon B. Weissman

Department of Computer Science and Engineering, University of Minnesota, Minneapolis MN, U.S.A.
{blee, jon}@cs.umn.edu

Abstract. Recently, there has been considerable interest in providing high performance codes as network services. In particular, high performance network services provide improved performance by executing complex and time-consuming applications (or part of an application) on remote high performance resources. However, since service providers resources are limited, without effective resource scheduling, end-users will not experience performance improvement. In this paper, we propose adaptive resource harvesting algorithms to schedule multiple concurrent service requests within network services. The preliminary results show that our approach can achieve service time improvement up to 40% for a prototypical parallel service.

1 Introduction

Recently, there has been considerable interest in providing high performance codes as network services. High performance applications such as data mining [4], theorem proving and logic [3], parallel numerical computation [2],[6] are example services that are all going on-line. Network services allow the end-users to focus on their applications and obtain remote services when needed simply by invoking remote services across the network. The primary advantages of using network services, in particular high performance network services, are:

- by executing complex and time-consuming applications (or part of an application) on remote sites that provide high performance resources, the end-users will experience significantly reduced service time.
- the end-users need not be involved with maintaining low-level infrastructure to run service codes because such activities are taken care of by the service providers.

Network services imply the potential of multiple concurrent users across the network. Moreover, since service providers resources are limited, effective resource management is essential to providing acceptable performance. To address this problem, our approach for scheduling multiple concurrent service requests within network services is based on ***Resource Harvesting***, where resources are dynamically added/removed to/from active service requests to support high performance.

The paper is organized as follow: Section 2 gives the related work and Section 3 describes the system model. Section 4 presents the resource harvesting algorithms and Section 5 shows the experimental results. Finally, we conclude in Section 6.

2 Related Work

Much research has been conducted on efficient resource management for executing high performance applications on multiple resources. However, most of them are limited to scheduling a single application using shared resources whereas in our model, the resource management system schedules multiple concurrent service requests to meet the performance objective.

[1] provides convenient tools for parametric computation experiments. Users can vary parameters related to the experiment by using a simple declarative parametric modeling language and the parametric engine manages and maintains the whole experiment. The scheduling scheme employed in the system is based on an economic model, which selects resources that meet the deadline and minimize the cost of computation. [2] and [6] are two representative network service infrastructures and they bear strong similarities both in motivation and general design. For each user request, a scheduler (or an agent in [2]) selects a set of servers that can handle the computation and ranks them based on the minimum completion time. [5] proposed local learning algorithms for the prediction of run-specific resource usage on the basis of run-time input parameters. Our run-time prediction technique is similar to their approach. However, they did not consider parallel applications. [7] proposed deadline scheduling that uses a load correction mechanism and a fallback mechanism to minimize the overall occurrence of deadline misses as well as their magnitudes. Unlike other works mentioned above, its goal is to schedule multi-client requests on multi-server environments.

In contrast to our work, none of these works support dynamic resource addition and removal to user requests. For example, once a server is allocated to the user request, it remains until the request completes or the server fails.

3 System Model for Network Services

We believe that some classes of high performance services are not appropriate for the resource harvesting technique. For example, network services with complex workflows may suffer from increased overhead for saving and redistributing their states when new resources are added in the middle of their execution. Therefore, the target classes of high performance network services that we are considering are data parallel and distributed services, which are common to high performance computing. Data parallel services are those that require communication between the constituent processors and the communication patterns are symmetric. Examples include numeric servers, N-body simulator, parallel CFD, etc. Distributed services are similar to data parallel services, but do not require communication among the distributed processes. Examples include stochastic simulation (e.g. monte carlo), parameter studies, etc.

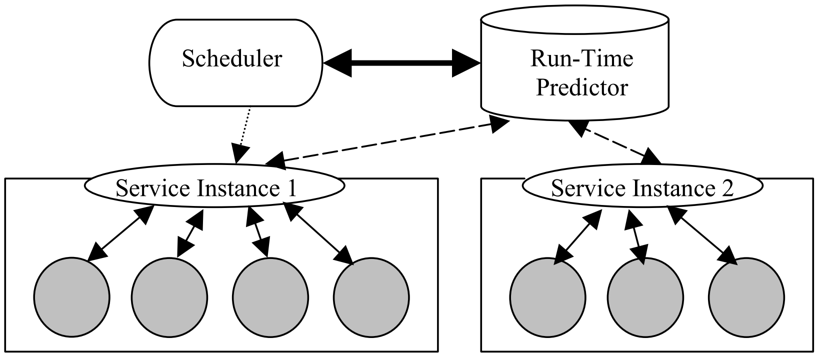

Fig. 1. General run-time infrastructure for network services: service instance 1 is using four resources and service instance 2 is using three resources.

Fig. 1 shows the general run-time infrastructure for network services. It consists of three primary components: scheduler, run-time predictor and service instances. The scheduler contacts the run-time predictor to acquire the estimated run-time of the service requests for resource scheduling. If it is needed to add/remove resources to/from active service instances, the scheduler contacts corresponding service instances, where adaptive actions are actually implemented. In addition, the service instances are responsible for reporting performance information to the run-time predictor so that the run-time predictor can maintain up-to-date performance information.

4 Adaptive Resource Management

Network services imply the potential of multiple concurrent users across the network and heterogeneous service requests (e.g. requests with different input parameters). Therefore, the run-time infrastructure must provide acceptable performance for a wide-spectrum of users. Our approach for scheduling multiple concurrent service requests is based on resource harvesting, where resources are dynamically added/removed to/from the active service requests to support high performance. For example, a scheduler may initially allocate a large number of resources to a request because there are no other requests pending. But when competing requests start to arrive and insufficient resources are available to run them, the scheduler may choose to harvest resources from running service requests. Therefore, performance prediction is crucial to adaptive resource management because deciding how to best allocate resources dynamically depends on the estimation of service execution time. The overhead of harvesting must be measured within the service and made available as part of the decision process. Resource harvesting raises two fundamental questions that should be addressed.

- *From which service requests should resources be harvested?*
- *How many resources should be harvested?*

One way to address the first question is the service provider establishes priority classes of its user base so that higher priority requests can harvest resources from lower priority requests. The second question is more complex because performance gain achieved by resource harvesting must be able to amortize the performance loss experienced by requests from which resources are taken away.

4.1 Performance Prediction

Performance prediction is needed by the scheduler to estimate the cost of executing a specific instance of a service request on a given amount of resources. The service execution time depends on not only the number of resources but also the input parameters to the service. For example, the time to solve NxN system of equations on K resources may be different from the time required to solve the system on L resources or to solve MxM system of equations on N resources. For simple services such as matrix multiplication, it is possible to predict run-time accurately using static cost functions. However, for complex services, we believe that static specification of detailed cost functions is not always possible. For the latter case, we use local linear regression and clustering technique to predict run-time, where local linear regression is applied using a subset of prior performance data that are clustered near the new data point. Performance history is organized in a two dimensional matrix, where each column represents the resource set and each row represents performance of the service given a set of input parameters on each resource set. To fill in the matrix, a configurable parameter, *cluster range*, is used. It determines whether or not two different input parameters can be regarded as similar in terms of the performance when the same resource set is used. For example, we might say that the run-time for 10000x10000 matrix multiplication is the same as the time for 10001x10001 matrix multiplication if the same resource set is used. When a triple < input parameter, resource set, performance> is known after finishing a service request, the corresponding cell in the matrix is located and the following condition is tested:

$$\alpha \leq \beta \pm (\beta \times cluster\,range)$$
$$\alpha : new\,performance\,data,\ \beta : the\,average\,value\,of\,the\,performance\,in\,the\,cell$$

If the condition is satisfied, then the new performance data is stored into the cell. Otherwise, a new row for the input parameters is created. Note that each cell can maintain several performance data with different input parameters values.

To predict the performance of the service given (input parameters: *i*, resource set: *j*), the scheduler first locates the corresponding cell in the performance history matrix indexed by (*i, j*). If it is populated, then it simply returns the average value of the performance data in the cell. Otherwise, it finds the nearest two cells on the same row as the cell and applies local linear regression using the performance data in the two cells (row-wise prediction). After that, it finds the nearest two cells on the same column as the cell and does the same operation (column-wise prediction). Finally, the average value of the two estimated values are returned as estimated run-time. Row-wise prediction reflects the performance change depending on resource set given the input parameters, whereas column-wise prediction reflects the performance change depending on input parameters given the same resource set.

4.2 Shortest-Remaining-Time Harvesting

The idea behind shortest-remaining-time harvesting (SRT_Harvest) is only when a new service request, *S*, can finish earlier than other service requests that are currently running, then *S* can harvest resources from those service requests to enable it to run. The behavior of the algorithm is determined by two configurable parameters:

- *HP (Harvesting Parameter)*: controls how aggressively the system can harvest resources from running requests.
- *WP (Wait Time Parameter)*: defines the maximum wait time threshold for each request. It is proportional to the minimum run-time of the request.

Fig. 2 describes the algorithm. If the run-time of a service request is long, its resources are frequently taken away for shorter requests, which sometimes results in the request not making any progress. To prevent the starvation of longer requests, whenever resources are available, the scheduler checks if there are any pending requests whose total wait time exceeds the maximum threshold, defined by *(EstimatedMimimumRunTime * WP)*. If so, resources are allocated to those requests and the resources are marked as "*Non-Harvestable*" so that no requests can take resources away from them (line 1).

Before harvesting resources, the scheduler contacts each of the active requests to acquire the current status information as to progress. For example, iterative services will return information on how many iterations has been done and the average iteration time. Using this information, the scheduler computes the available time of the resources that are currently being used by the active requests. Then, for each request in the wait queue, the scheduler computes the best performance achievable

```
SRT_Harvest () {
1.   Find pending requests of which wait time exceeds the maximum wait time
       threshold. Assign resources to those requests and mark the resources as "Non-
       Harvestable".
2.   Contact active service instances and compute the available time of resources that
       they are using.
3.   while (!done)    {
4.       Find a pending request whose run-time is smallest through resource
           harvesting.
5.       Start the request using selected resources; if the resources are being used by
           other requests, then send "resource removal" messages to them;
6.       If there is change in the wait queue due to resource harvesting, then go to 3.
7.   }
8.   for (each active request r that only a subset of resources are harvested) {
9.         Check if there is any pending request that can finish earlier than r using the
             remaining resources of r.
10.        If s is such request, assign the remaining resources of r to s and put r into the
             wait queue.
11.  }
}
```

Fig. 2. Pseudo-code for shortest-remaining-time harvesting algorithm.

using resource harvesting (line 3-7). It can harvest resources from request *R* for request *S*, only when the following condition is satisfied:

$$Run-Time\,of\ S\times HP<Remaining\,Time\,of\ R,\ (HP\geq 1.0)$$

This condition determines the candidate requests from which resources will be taken away and the number of resources from those requests. Thus, as long as a request *S* can finish earlier than *R*, *S* can harvest resources from *R*. If not, it can harvest none of the resources of *R*. If only a subset of resources of a request are harvested, then there could be requests in the wait queue that can finish earlier than the request using the remaining resource set. Since the algorithm favors shorter requests, in such case, the active request should relinquish its resources to the one in the wait queue. If there are multiple requests in the wait queue that satisfy the requirement, the one that can finish the earliest is selected (line 8-11).

4.3 Impact-Based Harvesting

In contrast to shortest-remaining-time harvesting, impact-based harvesting (IB_Harvest) focuses on resource lenders rather than resource borrowers (Fig. 3). For example, resources of an active service request, *R*, can be harvested for a new service request, *S*, only when the impact of resource harvesting that *R* will experience is below a threshold. The impact is defined in terms of service time. In order to compute the service time threshold for each request, we use a configurable parameter, *IP (Impact Parameter)* and the optimal run-time of the request. The optimal run-time is defined as the minimum run-time achievable.

If there are not enough resources available for a new request, for each active service request, the scheduler computes the number of resources that can be harvested from each of them (line 2-4). The number of harvestable resources of each active request is defined as follow:

```
IB_Harvest ()   {
1.    Contact active service instances and compute the available time of resources that
        they are using.
2.    for (each active service instance)    {
3.        compute the number of harvestable resources. If it is greater than zero, then
            mark the instance as “Harvestable”
4.    }
5.    m = the number of resources to harvest.
6.    while (m  > 0)   {
7.        for (i = 0; i < the number of harvestable instances; ++i)    {
8.            collect k resources from instance(i), where k is randomly generated.
9.            send “resource removal” message to instance(i).
10.           m = m – k;
11.       }
12.   }
}
```

Fig. 3. Pseudo-code for impact-based harvesting algorithm.

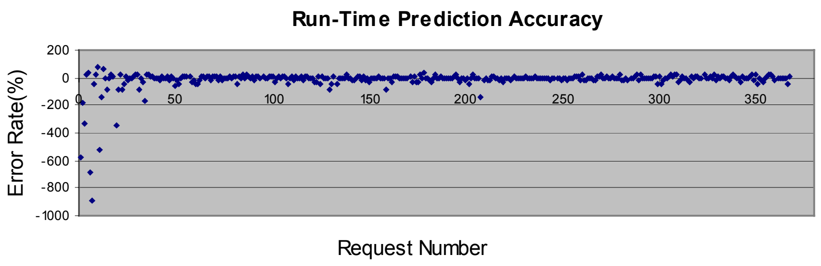

Fig. 4. Accuracy of Run-Time Prediction.

$$\max\{n : ElapsedTime + RemainingTime(n) < Service\ Time\ Threshold\}$$
$$ElaspedTime : Current\ Time - Service\ Start\ Time$$
$$RemainingTime(n) : estimated\ remaining\ run-time\ when\ n\ resources\ are\ used$$

The number of resources that will be allocated to the new request is the minimum number of resources with which it can finish earlier than the service time threshold. Once the harvestable service requests and the number of resources to harvest for the new request are determined, the scheduler collects resources randomly from each of the selected service requests until the desired number of resources is collected (line5-12).

When a service request finishes, it returns the harvested resources to service requests where the resources are collected. If the requests have already finished, instead of assigning the resources to active requests whose remaining time are small as in shortest-remaining-time harvesting, those resources are allocated to requests in the wait queue to reduce wait time of the requests.

5 Experimental Results

We have deployed an N-body simulation service to test the performance of the proposed scheduling policies. The objective of N-body simulation is to find the positions and movements of the bodies in space that are subject to gravitational forces from other bodies using Newtonian laws of physics [8].

In the prototype, the N-body simulation is implemented using Master/Slave paradigm, where the master maintain a bag of tasks and slaves repeatedly get tasks, update the bodies, then return the result to the master. Given *n* bodies and *p* slaves, the master divides the bodies into *m* blocks of size *n/p* and the slaves compute the forces between bodies in block *i* and those in block *j*. However, the computed forces do not reflect the effects of bodies in block *k (k != i, j)*, once the slaves finish computing forces of bodies in every pair of blocks, the master computes the total forces of each body. To use the N-body simulation service, the users submit four parameters: start time, end time, delta time (the length of the time interval), and input bodies. The first three parameters control the number of iterations. We deployed the prototype service on a Linux cluster consisting of 10 dual CPU PCs and the cluster is dedicated to the service.

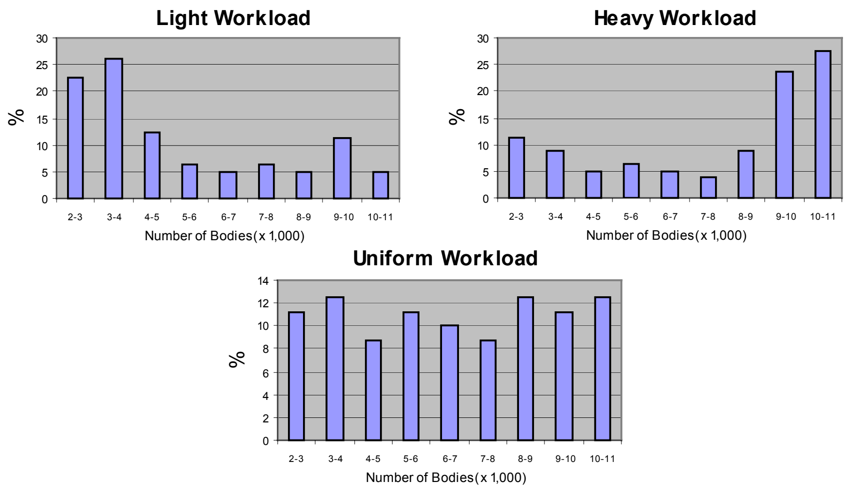

Fig. 5. Synthetic workloads.

Table 1. Configurable parameters for resource harvesting algorithms.

	Light workload	Uniform Workload	Heavy Workload
Cluster Range	0.05	0.05	0.05
HP	1.5	1.8	1.2
WP	12.0	8.0	12.0
IP	1.7	1.1	1.3

5.1 Performance Prediction

Accurate prediction of run-time is important in the resource harvesting algorithms we presented. For this experiment, we generated input parameters to the service randomly and cluster range is set to 0.05. The experimental results show that our prediction system can achieve estimation accuracy to within 4% (Fig. 4). Since initially there are not enough data in the performance history matrix, the error rates of the first few predictions are high. However, as clients requests are served, the prediction system learns the relationship among input parameters, resource set and the run-time. Therefore, after the learning phase, it can predict the run-time accurately.

5.2 Performance Comparison

To assess the performance of our scheduling policies, we generated three synthetic workloads: light workload, uniform workload and heavy workload (Fig. 5). The X-axis in the graph represents the number of bodies submitted to the service to compute movement. Note that since we fixed the three time-related parameters (start time, end

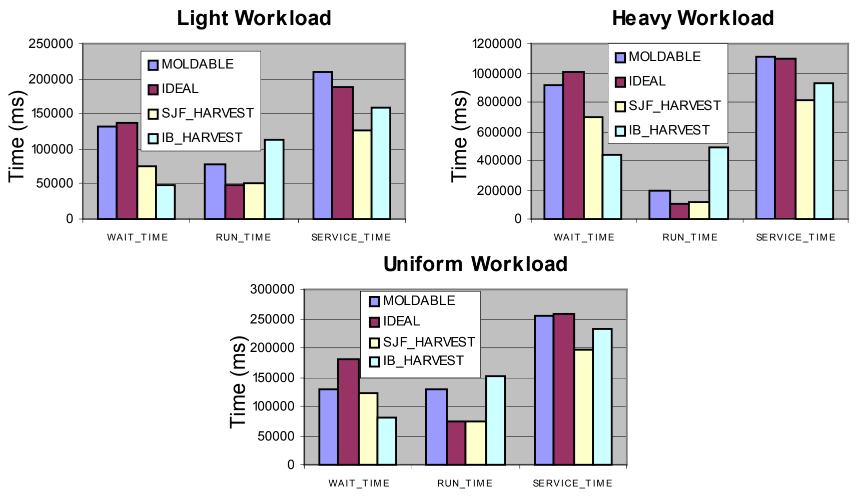

Fig. 6. Comparative performance for different resource scheduling policies.

time, delta time), which determine the number of iterations, the number of bodies controls the run-time. For example, the more bodies requested, the longer it takes to compute the result.

We compare performance against two simple scheduling policies: MOLDABLE and IDEAL (Fig. 6). MOLDABLE assigns idle resources up to the optimal number of resources for each request. If there are no available resources, the request is queued. Otherwise, the request is "molded" to the available number of resources. In IDEAL, only the optimal number of resources are assigned to the request. Therefore, if the number of available resources is less than the optimal number of resources, the request waits until the optimal number of resources are available. Both scheduling do not use resource harvesting. Table 1 shows configurable parameters of SRT_Harvest and IB_Harves used in the experiments.

For each workload, we measured average values of wait time, run-time, and service time of requests under different resource scheduling policies. Wait time of a request denotes a period of time in the wait queue, whereas run-time represents the time consumed to process the request. The service time is the sum of wait time and run-time. As IDEAL always waits until the optimal number of resources are available, its wait time is the highest but the average run-time of IDEAL is the smallest for the same reason. In SRT_Harvest, by executing shorter requests earlier than longer ones, the wait time is decreased significantly. In addition, because requests may not always run using the optimal number of resources, its average run-time is higher than that of IDEAL. However, if there is no shorter requests pending in the wait queue, instead of assigning idle resources to requests in the wait queue, it assigns as many resources as the optimal number of resources to the active requests. Therefore, the average run-time can be reduced. In this experiment, SRT_Harvest achieved service time improvement up to 40%, 27% and 20% for light workload, heavy workload and uniform workload, respectively. As in SRT_Harvest, IB_Harvest also dynamically

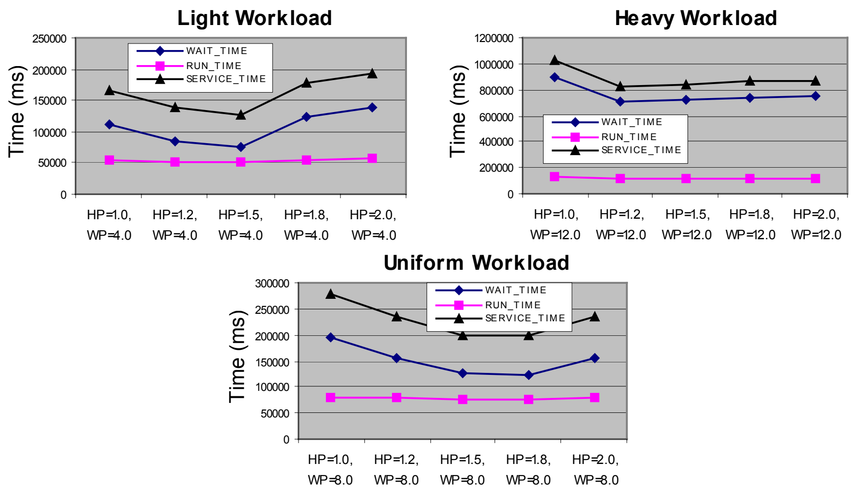

Fig. 7. Sensitivity to harvesting parameter.

collects resources for new requests whenever there are not enough resources for them. Therefore, its average wait time is also smaller than those of simple policies. Moreover, unlike SRT_Harvest, since IB_Harvest favors requests in the wait queue, the average wait time is even smaller than that of SRT_Harvest. However, due to resource harvesting, each request can use only the minimum number of resources, which leads to increased run-time.

5.3 Sensitivity to Configurable Parameters

In theory, running shortest requests first always reduces the average wait time. Therefore, in shortest-remaining-time harvesting, *HP=1.0* should provide the best performance. However, due to *WP*, shorter requests may wait until non-harvestable requests finish. Furthermore, smaller *HP* makes the wait time of longer requests reach the maximum wait time threshold faster because longer requests either may not be selected for execution or may relinquish all of their resources frequently to shorter requests. These two behaviors make the wait time of shorter requests longer if they arrive when non-harvestable requests are using all of the system resources (Fig. 7). However, as *HP* increases, the total wait time also increases because shorter requests may not be executed even though longer requests are using resources. The reason for choosing a larger value as *WP* for a heavy workload is as the run-time of each request is relatively high in the heavy workload, small *WP* makes the total wait time of each request exceed the maximum wait time threshold quickly. Therefore, it may not take advantage of resource harvesting. This behavior is explained in Fig. 8.

Fig. 8 shows that as *WP* increases, the average performance improves in both workloads. This is quite straightforward because with a very large *WP*, whenever shorter requests arrive, they acquire resources from longer requests. On the other

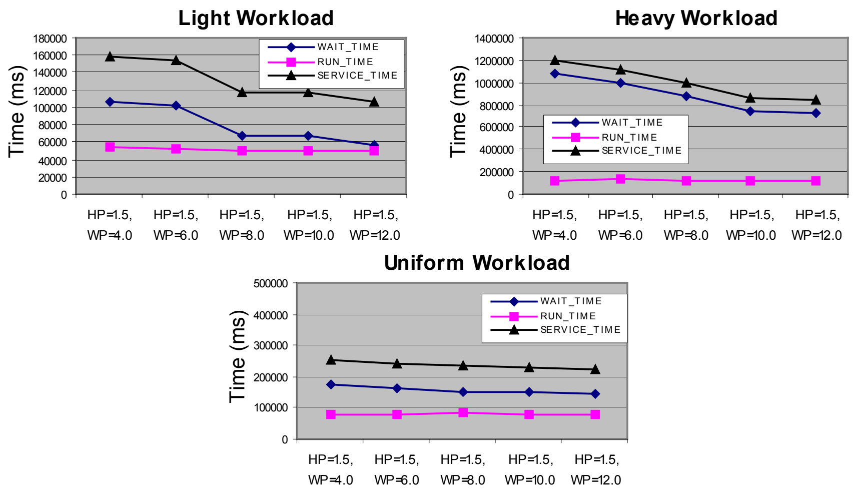

Fig. 8. Sensitivity to wait time parameter.

hand, if a small *WP* is used, as the maximum wait time threshold of each request becomes small, they become non-harvestable quickly. Therefore, even if shorter requests arrive, they may not acquire resources, which results in increased wait time.

Larger *IP* allows more frequent resource harvesting. Therefore, as *IP* increases, the average wait time decreases. However, at certain point, since most of the active requests are using the minimum number of resources, the decrements of average wait-times cannot compromise the increments of average run-times (Fig. 9). In addition, due to increased run-times, the available time of resources also increases, which

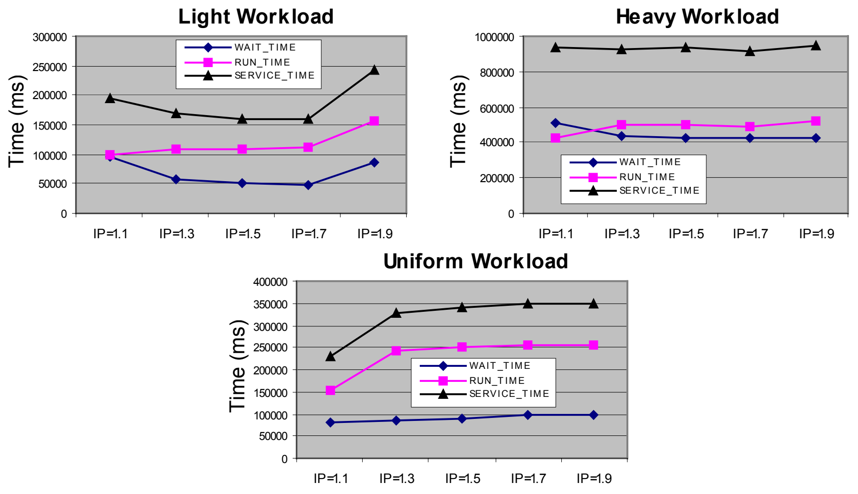

Fig. 9. Sensitivity to impact parameter.

results in increased average wait-times.

Acknowledgements
This work was sponsored in part by the Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAD19-01-20014

6 Conclusions

In this paper, we presented the adaptive resource scheduling technique to handle multiple concurrent service requests within network services. Novel aspect of our approach is resource harvesting, where resources are dynamically added/removed to/from active service requests. The preliminary results using N-body simulation service show that adaptive scheduling policies using resource harvesting can achieve significantly improved service time.

References

1. R. Buyya, D. Abramson, J. Giddy: Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid, Proceedings of 4th High Performance Computing in Asia-Pacific Region (2000)
2. H. Casanova, J. Dongarra: Netsolve: A Network Server for Solving Computational Science Problems, International Journal of Supercomputing Applications and High Performance Computing. Vol. 11(3). (1997)
3. D. Dill: SVC: The Standard Validity Checker, http://www.sprout.standford.edu/SVC
4. R.L. Grossman, S. Kasif, D. Mon, A. Ramu, B. Malhi: The Preliminary Desgin of Papyrus: A System for High Performance, Distributed Data Mining over Clusters, Meta-Clusters and Super-Clusters, Proceedings of KDD-98 Workshop on Distributed Data Mining (1998)
5. N. H. Kapadia, J. Fortes, C. Brodley: Predictive Application-Performance Modeling in a Computational Grid Environment, Proceedings of 8th International Symposium on High Performance Distributed Computing (1999)
6. H. Nakada, M. Sato, S. Sekiguchi: Design and Implementation of Ninf: Towards a Global Computing Infrastructure, Journal of Future Generation Systems, Metacomputing Issue (1999)
7. A. Takefusa, H. Casanova, S. Matsouka, F. Berman: A Study of Deadline Scheduling for Client-Server Systems on Computational Grid, Proceedings of 10th International Symposium on High Performance Distributed Computing (2001)
8. B. Wilkinson, M. Allen: Parallel Programming, Prentice Hall (1999)