GarQ: An Efficient Scheduling Data Structure for Advance Reservations of Grid Resources

Anthony Sulistio[1], Uros Cibej[2], Sushil K. Prasad[3], Borut Robic[2], and Rajkumar Buyya[1]

[1]GRIDS Laboratory
Dept. of Computer Sc. & Software Eng.
The University of Melbourne, Australia
{anthony, raj}@csse.unimelb.edu.au
Phone: +61 3 8344 1344
Fax: +61 3 9348 1184

[2]Fac. of Computer & Information Sc.
The University of Ljubljana
25 Trzaska, 1000 Ljubljana, Slovenia
{uros.cibej, borut.robic}@fri.uni-lj.si
Phone: +1 404 413 5731
Fax: +1 404 413 5717

[3]Dept. of Computer Science
Georgia State University
Atlanta, USA
sprasad@cs.gsu.edu
Phone: +386 1 4768 867
Fax: +386 1 4264 647

Abstract

In Grids, users may require assurance for completing their jobs on shared resources. Such guarantees can only be provided by reserving resources in advance. However, if many reservation requests arrive at a resource simultaneously, the overhead of providing such service due to adding, deleting, and searching, will be significant. An efficient data structure for managing these reservations plays an important role in order to minimize the time required for searching available resources, adding, and deleting reservations. In this paper, we present new approaches to advance reservation in order to deal with the limitations of the existing data structures, such as Segment Tree and Calendar Queue in similar problems. We propose a Grid advanced reservation Queue (GarQ), which is a new data structure that improves some weaknesses of the aforementioned data structures. We demonstrate the superiority of the proposed structure by conducting a detailed performance evaluation on real workload traces.

Keywords: data structure, advance reservation, segment tree, calendar queue, and grid computing.

1 Introduction

Grids [11] and peer-to-peer (P2P) [15] network technologies enable the aggregation of distributed resources for solving large-scale and computationally-intensive applications. These technologies are well-suited for Bag-of-Tasks (BoT) applications [8], wherein each application consists of independent tasks or jobs [1]. Some projects such as Nimrod-G [6], Gridbus Broker [20] and SETI@home [2] utilize these technologies to schedule compute-intensive parameter-sweep applications on available resources [17].

Managing various resources and applications scheduling in highly dynamic Grid environments is a complex and challenging process. Resource management is not only about scheduling large and compute-intensive applications, but also the manner in which resources are allocated, assigned, and

[1]Corresponding author: raj@csse.unimelb.edu.au

accessed. In most systems, submitted jobs are initially placed into a queue if there are no available resources. Therefore, there is no guarantee as to when these jobs will be executed. This causes problems in time-critical or parallel applications, such as task graph, where jobs may have interdependencies.

Advance Reservation (AR) is the process of requesting resources for use at specific times in the future [16]. Common resources that can be reserved are compute nodes and network bandwidth. AR in a scheduling system solves the above problem by allowing users to gain *simultaneous* and *concurrent access* to adequate resources for the execution of such applications [19]. Currently, several Grid systems are able to provide AR functionalities, such as GARA [12], and ICENI [14].

In order to reserve the available resources in such AR systems, a user must first submit a request by specifying a series of parameters such as number of compute nodes needed, and start time and duration of his/her jobs [13]. Then, the AR system checks for the feasibility of this request. If there are no available nodes for this requested time period, the request is rejected. Consequently, the user may resubmit a new request with a different start time and/or duration until available nodes can be found. Given this scenario, the choice of efficient data structure can significantly minimize the time complexity needed to search available compute nodes, add new requests, and delete existing reservations. Moreover, a well-designed data structure provides the flexibility and easiness in implementing various algorithms.

Some data structures are tailored to specific applications. For example, a tree-based data structure is commonly used for admission control in network bandwidth reservation [3, 21, 23]. Each tree node contains a time interval and the amount of reserved bandwidth in its subtree. Therefore, a leaf node has the smallest time interval compared to its ancestor nodes. Hence, the number of bandwidth required for a single reservation is stored into one or more fitting nodes. In general, a tree-based structure has a time complexity of $O(log\ n)$ for searching the available bandwidth, where n is the number of tree nodes. This approach is considered to be better than using a sorted Linked-List data structure [22], which has a sequential searching method leading to $O(tot_{AR})$ time complexity, where tot_{AR} is the total number of reservations. This is because the List does not partition each reservation into a fixed time interval like a tree-based structure. Contrarily, a study done by Burchard [5] found that arrays provide better performance than a tree-based structure, such as a Segment Tree [3], for processing new requests and searching larger time intervals. The study was conducted to measure the admission speed of a bandwidth broker using each structure in a multilink admission control environment.

The previous studies are primarily focused on finding out the search time of the aforementioned data structures. However, these studies do not explicitly consider *add* and *delete* operations for adding new requests and deleting existing reservations respectively, for these data structures. This is because, for reserving network bandwidth, each tree node and index in Segment Tree and Array respectively, only stores information regarding the allocated reserved bandwidth. Hence, the performance of addition and deletion can be neglected. In contrast, a data structure needs to keep additional information for reserving compute nodes in Grid systems, such as user's jobs for executing on the reserved nodes, and their status for monitoring purposes. In addition, most of these studies, except done by [5], do not consider an *interval search* operation, where the data structure finds an alternative time for a rejected request. This operation helps users who requests got rejected in negotiating a suitable reservation time. Therefore, the performance of this operation also needs to be considered when choosing the appropriate data structure.

The contributions of this paper are as follows. We first describe modified versions of Linked List and Segment Tree data structures to support *add*, *delete*, and *search*, as well as the *interval search* operation capable of dealing with advance reservations in computational Grids. For this, we had to specifically develop an algorithm for finding closest interval to a requested reservation for Segment Tree. Second,

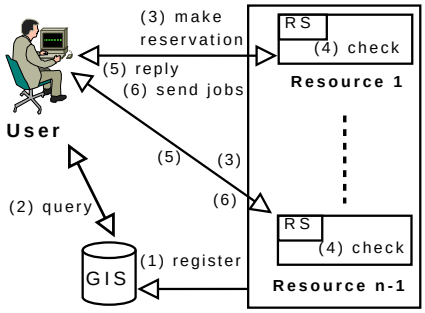

Figure 1. An overview of the Grid model.

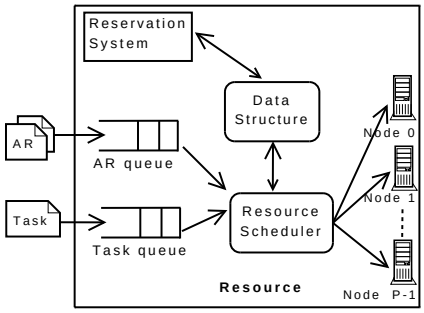

Figure 2. Overall resource model that supports reservation.

we introduce and adapt Calendar Queue [4] data structure for managing reservations as well. Calendar Queue is a priority queue for future event set (FES) problems in discrete event simulation. FES shares similar characteristics to advance reservations in Grids, where it records future events, and schedules them in chronological order. Next, we propose a new data structure that is tailored to handle the above operations efficiently. The new data structure is called Grid advanced reservation Queue (GarQ), which is a combination of Calendar Queue and Segment Tree, for administering reservations efficiently for the above operations. We demonstrate this by doing a comprehensive performance evaluation using several real-world workload traces from the Parallel Workload Archive [10]. The results show that GarQ has a better performance time on average when dealing with reservation operations compared to other data structures.

The rest of this paper is organized as follows. Section 2 describes the overall model, whereas Section 3 describes each modified data structure. Section 4 explains our proposed structure, and Section 5 evaluates each data structure's effectiveness on real workload traces. Finally, Section 6 concludes the paper and presents future work.

2 Description of the Model

In our model, as depicted in Figure 1, each resource has a Reservation System (RS), which is responsible for handling reservation queries and requests. Also in the model, a Grid consists of a Grid Information Service (GIS), resources and users. The interaction among these components are mentioned as follows. Each resource advertises its availability to a GIS (step 1). A user queries a list of available resources to the GIS (step 2). In order to reserve one or more compute nodes (CNs), a user needs to submit a reservation request to a resource (step 3). In this paper, the request is defined as $reserv(t_s,\ t_e,\ numCN)$, where t_s denotes the reservation starting time, t_e denotes the reservation ending time, and $numCN$ indicates the number of compute nodes to be reserved respectively. When a resource receives the request, it checks for availability (step 4). More specifically, the RS asks the data structure for this request, as shown in Figure 2. Then, the resource replies back to the user whether it can accept the request or not (step 5). If the request has been accepted, then the user sends his/her jobs (step 6) or goes back to (step 3) with a new reservation request with a different time interval.

Figure 2 also shows the open queueing network model of a resource applied to our work. In this model, there are two queues: one is reserved for AR jobs while the other one is for parallel and independent

jobs. Each queue has a finite buffer with size S to store objects waiting to be processed by one of P independent CNs. All CNs are connected by a high-speed network. The CNs in the resource can be homogeneous or heterogeneous. In this paper, we assume that a resource has homogeneous CNs, each having the same processing power, memory and harddisk. Therefore, the primary role of the data structure is to store and to update the information about CNs' availability as time progresses. Then, a resource scheduler is responsible for managing incoming jobs and assigning them to available CNs.

The reservation model, as shown in Figure 1, uses a two-phase commit and gives the user a chance to negotiate with the RS if the request gets rejected [13]. Hence, in order to support this reservation scenario, a data structure needs to perform the following basic operations:

- *search*: checking for availability of CNs in a given time interval. This operation is defined as $searchReserv(t_s,\ t_e,\ numCN)$.
- *add*: inserting a new reservation request into the data structure. This operation is performed only when the previous search phase succeeded. For addition, the new reservation is represented as $addReserv(t_s,\ t_e,\ numCN,\ user)$, where $user$ is an object storing the user's jobs and other relevant information.
- *delete*: removing the existing reservation from the data structure. This operation is conducted only when the add phase succeeds and the reservation's finish time has passed. It is described as $deleteReserv(t_s,\ t_e,\ numCN)$.

In addition to the above basic operations, an *interval search* operation is required for searching for the next available start time within a given time interval. Therefore, if a request is rejected, the RS can suggest an alternative starting time. This operation is represented as $suggestInterval(t_s,\ t_e,\ numCN)$.

Figure 3 shows an example of existing reservations represented in a time-space diagram. When a new request from *User5* arrives, the RS checks for any available compute nodes. In this case, the required number of nodes is 2. However, only one node is available, hence, this request will be rejected. By performing $suggestInterval(11,\ 13,\ 2)$ on this request, the RS manages to find the next available time, which is from time 13 to time 15.

3 Adapting Existing Data Structures

In general, a data structure that deals with a resource reservation can be categorized into two types, i.e. a *time-slotted* structure and a *continuous* one. A time-slotted data structure divides the reservation time into slots based on a fixed time interval. For example, 1 slot may represent 1 second or 5 minutes or 1 hour of a CN's computation time. Hence, the start time and duration time of a reservation will be partitioned, compared with the existing ones and placed to the appropriate slots (if accepted). Examples of this type of data structure are Segment Tree and Calendar Queue, and they will be discussed next. In contrast, a continuous data structure, such as Linked List [22] is more flexible. Therefore, it allows a reservation to start or finish at arbitrary times. Moreover, it obviates the need to have a minimum duration time for each reservation as compared to a time-slotted structure.

3.1 Segment Tree

Segment Tree, as shown in Figure 4, is a binary tree where each node represents a semi-open time interval $(X, Y]$. The left sibling of the node represents the interval $\left(X, \frac{X+Y}{2}\right]$, and the right sibling

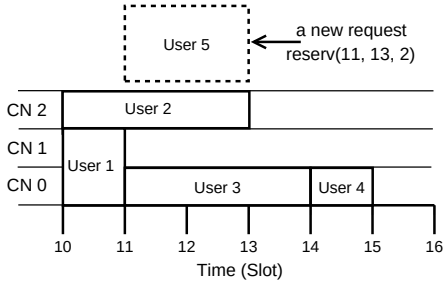

Figure 3. An example of advance reservations for reserving compute nodes. The maximum available compute node is 3. A dotted box denotes a new request.

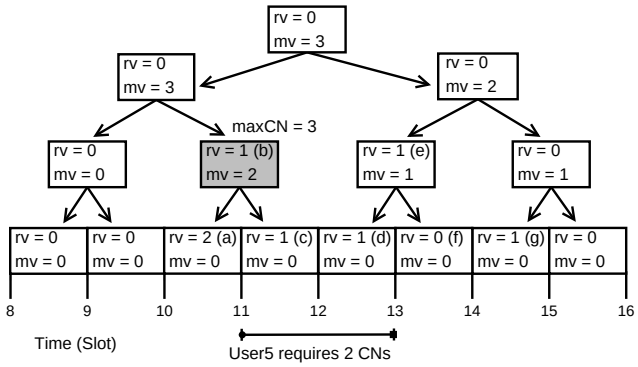

Figure 4. A representation of storing reservations in Segment Tree. A request from $User5$ is rejected, because node (b), representing a time interval $(10, 12]$, uses all the compute nodes.

represents the interval $\left(\frac{X+Y}{2}, Y\right]$. Each node has also the following information:

- rv: the number of reserved CNs over the entire interval. When a reservation which spans the entire interval $(X, Y]$ is added, rv is increased by the number of CNs required by this reservation. No further descent into the child nodes is needed.
- mv: the maximum number of reserved CNs in the child nodes. In the leaf nodes, the mv value is 0. The total number of reserved CNs in the interval of a leaf node is the sum of all rv of nodes on the path from the root node to the leaf node.

Note that the complete tree in Figure 4 is not drawn here due to lack of space. However, the height of Segment Tree can be computed as:

$$height = \log_2 \left(\frac{interval\_length}{slot\_length} \right) \tag{1}$$

where $interval\_length$ is the length of the whole interval we want to cover, and $slot\_length$ is the length of the smallest time slot. In our implementation, $interval\_length$ is one month (30 days), and the leaves of this tree represent $slot\_length$ of 5 minutes. To deal with reservations for an arbitrary time T, we first compute a new time which fits into this interval. In order not to overlap reservation from different months, we assume that no reservations are made more than one month in advance. This assumption is also valid for other data structures. As a result, the whole tree can be reused for the next month interval. Hence, the tree is only going to be built once in the beginning.

All operations on Segment Tree are performed recursively. Before giving a brief description of the operations, we define some common notations that will be used, as follows:

- N is the node the recursion is currently in with N_l is the left sibling and N_r is the right sibling.
- $(l_N,\ r_N]$ is the interval of the node N.

- $(l,\ r,\ numCN)$ is the input to all the operations.
- $maxCN$ is the maximum number of available CNs in the system.

For the *search* operation, if a reservation request covers the entire interval of the current node, such that $(l,\ r] == (l_N,\ r_N]$ && $(rv + mv + numCN) \leq maxCN$, then we have found enough free CNs and can terminate the recursion, as shown in Figure 4. Hence, Segment Tree is able to search quickly without having to go down to the leaf nodes for a larger interval.

Likewise, for the *add* operation, if $(l,\ r] == (l_N,\ r_N]$, then we increase rv by $numCN$ and return $(rv + mv)$ to the parent node. Figure 4 shows how the reservations are added into the tree. By using Figure 3 as an example, *User1* is stored into node (a), *User2* to node (b) and (d), *User3* to node (c) and (e), and *User4* to node (g). Moreover, the values of rv and mv on each node are updated accordingly. Removing a reservation is very similar to adding one, so the description can be omitted from this paper.

Algorithm 1: $suggestInterval(l,\ r,\ numCN)$ in Segment Tree

```
1  if numCN > N_availableCN then return −1;
2  if (l, r] == (l_N, r_N] then return l_N;
3  else
4      if N is a leaf node then return l;
5      if l ∈ (l_{N_l}, r_{N_l}] and r ∈ (l_{N_r}, r_{N_r}] then
6          leftS ← N_l.suggestInterval(l, l_{N_l}, numCN);
7          if leftS == l then
8              rightS ← N_r.suggestInterval(l_{N_r}, l_{N_r} + Δ − (l − l_{N_r}), numCN);
9              if rightS == l_{N_r} then return l;
10             else return rightS;
11         else return N.suggestInterval(leftS, leftS + Δ, numCN);
12     else if r ∉ (l_{N_r}, r_{N_r}] then
13         leftS ← N_l.suggestInterval(l, l_{N_l}, numCN);
14         if leftS == l then return l;
15         else return N.suggestInterval(leftS, leftS + Δ, numCN);
16     else return N_r.suggestInterval(l, r, numCN);
17 end
```

Searching for a free slot. Brodnik et. al. [3] do not describe the operation of finding a new free interval, closest to the proposed reservation $reserv(l,\ r,\ numCN)$, so we give a more detailed description of the implementation of this function. We have to point out that the operation described below finds the closest interval later than the current proposed interval. The description is given in pseudocode in Algorithm 1, and uses the common symbols defined as:

- $N_{availableCN}$ is number of available CNs in the whole interval of the node N;
- $leftS,\ rightS$ are temporary variables, that store the suggested starting time from the left and right subtree respectively; and
- Δ is the length of the reservation interval, so simply $\Delta = r - l$.

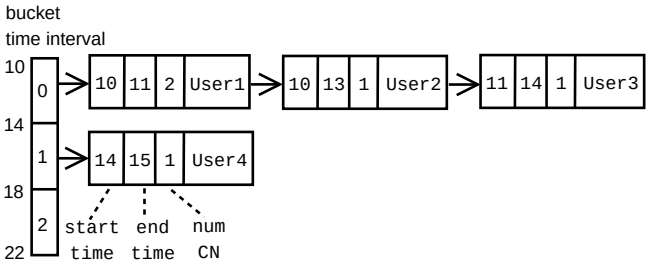

Figure 5. A representation of storing reservations in Calendar Queue, with δ = 4.

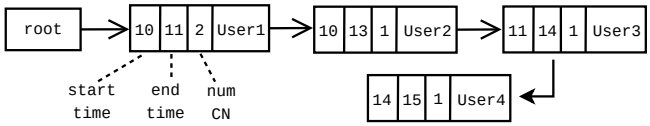

Figure 6. A representation of storing reservations in Linked List.

The function recursively searches for a suitable interval. In the case where the reservation interval covers the whole interval of the current node N, it examines the number of available CNs in this interval (lines 1–2). If there are enough CNs, the function returns the leftmost point of the interval l_N, and the rightmost point r_N, otherwise. When the searched interval does not cover the entire interval of the current node (lines 3–17), the function deals with four different possibilities:

1. Current node is a leaf (line 4). This is the boundary condition where the interval is a candidate for the free slot.

2. The interval $(l,\ r]$ covers the intervals of both the node N_l and N_r (lines 5–11). First it finds a candidate interval in the left sibling ($leftS$). If the suggested interval is equal to the original interval (starting at l) we can check if there is enough space in the right subtree as well. Otherwise we re-check the interval $(l_N,\ r_N]$ with a new proposed interval $(leftS,\ leftS + \Delta]$.

3. The interval $(l,\ r]$ covers only the interval of the node N_l (lines 12–15). Similarly to the approach in the first case (above), the procedure searches the left subtree. If the suggested interval is the same as the proposed one, we return it, otherwise we re-check the interval $(l_N,\ r_N]$.

4. The interval $(l,\ r]$ covers only the interval of the node N_r (line 16). In this case we recursively search for a free slot only in the right subtree.

In the case where there is no free interval in Segment Tree, the function returns -1.

3.2 Calendar Queue

Calendar Queue (CalendarQ) was introduced by Brown [4], as a priority queue for future event set problems in discrete event simulation. It is modeled after a desk calendar, where each day or page contains sorted events to be scheduled on that period of time. Hence, CalendarQ is represented as one or more pages or "bucket" with a fixed time interval or width δ. Then, each bucket contains a sorted linked list storing future events. Figure 5 shows how reservations are stored in CalendarQ with δ = 4 time interval, by using the example described in Figure 3. If a reservation requires more than δ, this reservation will also be duplicated into the next buckets. This approach makes the *search* operation easier since it only searches for a list inside each bucket.

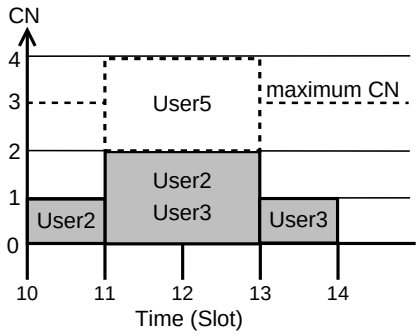

Figure 7. A histogram for searching the available CNs in Linked List for $User5$. A dotted box means a new request.

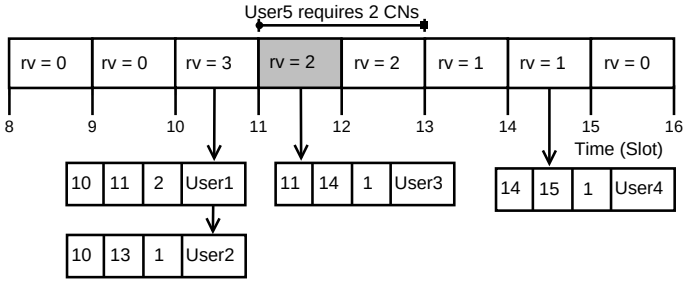

Figure 8. A representation of storing reservations in GarQ with Sorted Queue and δ = 1. A request from $User5$ is rejected because not enough CNs for slot $[11, 12)$ as shown by the shaded box.

In our implementation, we opted for a static CalendarQ where the number of buckets M and δ are fixed. Hence, these parameters do not need to be adjusted periodically as the queue grows and shrinks. Therefore, by choosing the proper settings for M and δ, CalendarQ performs constant expected time per event processed [9]. In addition, with the static approach, the whole CalendarQ can be reused for the next time period, similar to Segment Tree.

Overall, CalendarQ has a complexity of $O(k)$ for adding reservations, where k is the number of reservations in the list for each bucket. Deleting reservations require a fast $O(1)$ because they are sorted in the list, and CalendarQ only removes the reservations in the current bucket as time progresses. Searching for available CNs require $O(k\ m_{sub})$, where m_{sub} is the number of buckets within a subinterval.

3.3 Linked List

Linked List is the simplest and most flexible data structure of all, because accepted reservations will be inserted into the list based on their starting time. In Linked List, each node contains a tuple $\langle t_s,\ t_e,\ numCN,\ user\rangle$. Figure 6 shows how these reservations are stored by using the example as described in Figure 3.

Searching for available CNs. For a *search* operation, the implementation in Linked List is as follows. First, the List needs to find out which nodes have already reserved CNs within $[t_s,\ t_e]$ of the new request. By using the example as shown in Figure 3, we find that only *User2* and *User3* reserve these CNs within the time interval of *User5*. Second, it creates a temporary array for storing the number of CNs used within each time slot, including the new request as shown in Figure 7. Finally, it checks each time slot for sufficient member of available CNs. Therefore, for the *search* operation, Linked List has $O(tot_{AR}\ m_{sub})$, where tot_{AR} is the total number of reservations, and m_{sub} is the number of slots in the subinterval. The same approach also applied to the *interval search* operation, but shifting the time interval to $[t_s{+}\lambda,\ t_e{+}\lambda]$ instead, where λ is the length of busy period found from the previous *search* operation. The *interval search* operation ends when it reaches the tail of the List and/or $(t_e + \lambda)\ >\ (t_s + MAX\_LIMIT)$, where MAX_LIMIT denotes the maximum time needed for searching.

Adding and Deleting a reservation. These operations can be performed easily in Linked List by iterating through the list from the root node, and comparing each existing node based on its t_s. If the correct position or node has been found, then addition or deletion can be done respectively. Overall, List has $O(tot_{AR})$ complexity for *add* and *delete* operations. However, Linked List can become very inefficient for running these operations on many short reservations, because it needs to find the correct position or node starting from the root node.

4 The Proposed Data Structure: Grid Advanced Reservation Queue (GarQ)

After analyzing the characteristics of the modified Segment Tree and Calendar Queue data structures in the previous section, we propose an array-based structure for managing reservations in Grid computing. The idea behind this data structure was partially inspired by Calendar Queue and Segment Tree. By combining Calendar Queue and Segment Tree into this structure, we gained the following advantages:

- ability to add new reservations directly into a particular bucket. Hence, it has a fast $O(1)$ access to the bucket;
- ability to reuse these buckets for the next time period;
- built only once in the beginning;
- easy to search and compare by using iteration;
- easy to implement in comparison to Segment Tree and Calendar Queue; and
- flexibility in handling resource availability. In Grids, CNs can be added or removed periodically. This issue can be addressed by a reservation system or a resource scheduler by setting the amount of available CNs on that resource appropriately. Moreover, existing reservations can be relocated to other CNs through the *add* and *delete* operations.

The proposed data structure has buckets with a fixed δ, which represents the smallest slot duration, as with the Calendar Queue. Each bucket contains rv (the number of already reserved CNs in this bucket) and a linked list (sorted or unsorted), containing the reservations which start in this time bucket. Figure 8 shows how reservations are stored in "GarQ with Sorted Queue" with a δ = 1 time interval, by using the example described in Figure 3. For enabling a fast $O(1)$ access to a particular bucket, we use the following formula:

$$i = \left\lceil \frac{t}{\delta} \right\rceil \mod M \tag{2}$$

where i is the bucket index, t is the request time (in minutes), and M is the number of buckets in the data structure.

In what follows, we give a detailed description of the four operations: searching for available CNs, adding a reservation, deleting a reservation and searching for the closest free interval. Throughout the description of these operations, a common input for all of them is the tuple $\langle t_s,\ t_e,\ numCN \rangle$. Moreover, they use $start\_bucket$ and end_bucket, which denote the index of the start and end bucket in the reservation interval respectively. To determine the exact index, $get\_bucket\_index()$ function uses equation 2. We also use $maxCN$ to indicate the maximum number of CNs available in the system.

Algorithm 2: $searchReserv(t_s,\ t_e,\ numCN)$ in GarQ

```
1 start_bucket ← get_bucket_index(t_s);          /* get the starting index */
2 end_bucket ← get_bucket_index(t_e);            /* get the ending index */
3 for i = start_bucket to end_bucket do
4 |   if i == M then i ← 0;                      /* wrapping the array */
5 |   if bucket[i].rv + numCN > maxCN then return false;   /* slot is full */
6 end
7 return true;
```

Searching for available CNs. With GarQ, searching for available CNs is done by iterating through the entire interval and checking each bucket for free CNs, as shown in Algorithm 2. When i points to the end of the array or M, then it needs to search from the beginning of the array (line 4). Overall, the complexity of GarQ for searching is $O(m_{sub})$, where m_{sub} is the number of buckets within a subinterval.

Algorithm 3: $addReserv(t_s,\ t_e,\ numCN,\ user)$ in GarQ

```
1 start_bucket ← get_bucket_index(t_s);          /* get the starting index */
2 end_bucket ← get_bucket_index(t_e);            /* get the ending index */
3 bucket[start_bucket].addInfo(user);    /* store user's jobs & other details */
4 for i = start_bucket to end_bucket do
5 |   if i == M then i ← 0;                      /* wrapping the array */
6 |   bucket[i].rv += numCN;                     /* increase rv */
7 end
```

Adding a reservation. We assume there are enough CNs to add this reservation, i.e. a search has been done beforehand. Adding a new reservation is very similar to searching, and it is described in Algorithm 3. Hence, the complexity of our structure for addition is $O(m_{sub})$ or $O(k + m_{sub})$ when using unsorted and sorted queue respectively, where k is the number of reservations in a bucket list.

Deleting a reservation. Deleting an existing reservation applies to the same principle as adding a new one. It can be done by removing the reservation from the starting bucket and decrementing rv through out the relevant bucket interval.

Searching for a free slot. Searching for the closest interval is also straightforward in GarQ, as shown in Algorithm 4. This algorithm is similar to Algorithm 2, but the search interval is now expanded by MAX_LIMIT. This constant variable denotes the maximum time needed for the *interval search* operation, hence, it prevents the algorithm from searching the array infinitely. During the searching, a temporary counter $count$ indicates how many buckets still need to be searched (and have enough free CNs) before the operation can finish (line 9–13). At the end of the operation, the index of a new start bucket, new_start, is converted into the new starting time by using $convert\_index()$ function.

After describing these data structures, a summary of each of them is given in Table 1, including our proposed data structure, namely GarQ with either Unsorted or Sorted Queue. In the next section, we

Algorithm 4: $suggestInterval(t_s,\ t_e,\ numCN)$ in GarQ

```
1  start_bucket ← get_bucket_index(t_s);                    /* get the starting index */
2  end_bucket ← get_bucket_index(t_e);                      /* get the ending index */
3  tot_req ← 1 + end_bucket − start_bucket;                 /* total slots required */
4  new_start ← start_bucket;                                /* the new starting index */
5  count ← 0;                          /* count number of slots available so far */
6  finish ← get_bucket_index(t_s + MAX_LIMIT);              /* the last bucket to search */
7  for i = start_bucket to finish do
8      if i == M then i ← 0;                                /* wrapping the array */
9      if bucket[i].rv + numCN > maxCN then
10         new_start ← i + 1;                               /* points to the next bucket */
11         count ← 0;                                       /* reset the counter to zero */
12     else count ← count + 1;
13     if count >= tot_req then break;                      /* exit loop if found enough slots */
14 end
15 if count < tot_req then new_start ← (−1);                /* all slots do not have enough CNs */
16 new_time ← convert_index(new_start);                     /* convert bucket index into start time */
17 return new_time;
```

evaluate the performance of our data structure with the existing ones. We conduct the evaluation using real workload traces.

5 Performance Evaluation

In order to evaluate the performance of our proposed data structure, i.e. GarQ with Unsorted Queue (GarQ-U) and GarQ with Sorted Queue (GarQ-S), we compare them to Linked List (List), Segment Tree (Tree) with $slot\_length$ = 5 minutes, and static Calendar Queue (SCQ) with δ = 1 hour. For SCQ to be optimal, we choose the value of δ based on the jobs' average duration time as stated in Table 2. For GarQ-U and GarQ-S, we set each slot to be a 5-minute interval. All, except List, have a fixed interval

Table 1. Summary of the data structures, where n is the number of tree nodes, k is the number of reservations in the list for each bucket, m_{sub} is the number of buckets or slots within a subinterval, and tot_{AR} is the total number of reservations.

Name	Time Complexity		
	Add	Delete	Search
Segment Tree	$O(log\ n)$	$O(log\ n)$	$O(log\ n)$
Calendar Queue	$O(k)$	$O(1)$	$O(k\ m_{sub})$
Linked List	$O(tot_{AR})$	$O(tot_{AR})$	$O(tot_{AR}\ m_{sub})$
GarQ with Unsorted Queue	$O(m_{sub})$	$O(k + m_{sub})$	$O(m_{sub})$
GarQ with Sorted Queue	$O(k + m_{sub})$	$O(m_{sub})$	$O(m_{sub})$

Table 2. Workload traces used in this experiment.

Trace Name	Location	# Jobs	Mean Job Time	From	To
DAS2 fs0	Vrije Univ., Amsterdam	225,711	11.74 minutes	Jan 2003	Dec 2003
LPC EGEE	Clermont-Ferrand, France	242,695	52.07 minutes	Aug 2004	May 2005
SDSC BLUE	San Diego, USA	243,314	69.34 minutes	Apr 2000	Jan 2003

length of 30 days, as mentioned previously.

For the evaluation, we are investigating: *(i)* the total number of nodes or slots accessed throughout for each of the operations, including temporary ones for List and SCQ; *(ii)* the average runtime for using the above operations; and *(iii)* the average memory consumption for these data structures. Note that we conduct the experiment this way because we want to get a clear picture on how each data structure performs, without the interference of scheduling issues such as deadline, backfilling and job preemption. However, we will consider the affects of these issues when integrating the data structures with a resource scheduler as part of the future work.

We carried out the performance evaluation of these data structures by using simulation, because we need to conduct *repeatable* and *controlled* experiments that would otherwise be difficult to perform in real Grid testbeds. Therefore, we use GridSim toolkit [7, 18] in these experiments by simulating a cluster of 64 compute nodes, i.e. $maxCN = 64$.

5.1 Experimental Setup

We selected three workload traces from the Parallel Workload Archive [10] for our experiment, as summarized in Table 2. These traces were chosen because they represent a large number of jobs and contain a mixture of single and parallel jobs. In addition, the LPC trace was based on recorded activities from the EGEE (Enabling Grids for E-science in Europe) project, hence it is very suitable in conducting the evaluation. Moreover, as shown in Table 2, the average job duration time varies from 11 to 70 minutes. Hence, we can analyze in more detail the performance of each data structure for jobs with a short, medium and long duration time.

Although these traces were taken from the real production systems, the jobs' starting times were logged in increasing order. Hence, it might not be suitable for testing out the *interval search* operation. Therefore, we *shuffled* or *randomized* the starting time order of jobs for every 2-week period of each trace. Overall, we have 6 traces in this experiment: the 3 original ones and 3 shuffled ones. Several modifications have also been made to these traces, as mentioned below:

- If a job requires more than the total CNs of a resource, we set this job to $maxCN$.
- A request's starting time is rounding up to the nearest time interval. For example, if a job requests to start at time 01:03:05 (hh:mm:ss), then it will be moved to time 01:05:00.
- A job duration time is within [5 minutes, 28 days]. We limit the maximum duration time to prevent overlapping reservations from different months. Hence, each structure, except for Linked List, can be reused and built only once.

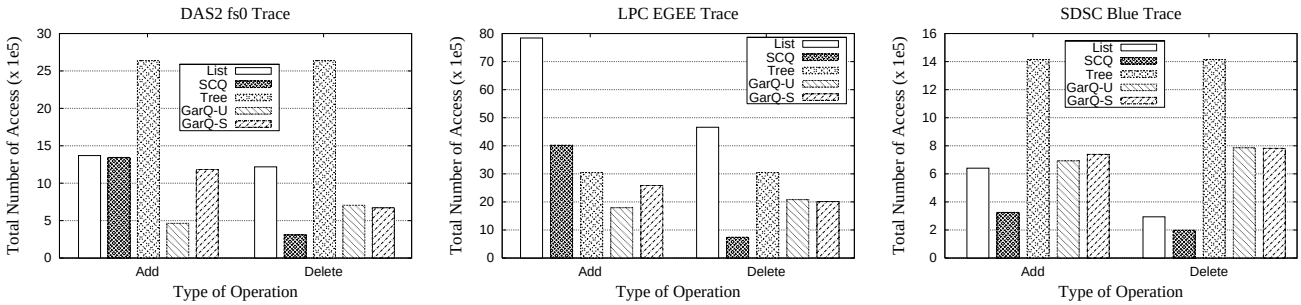

Figure 9. Total number of nodes accessed during *add* and *delete* operation using original traces (shorter bars are better).

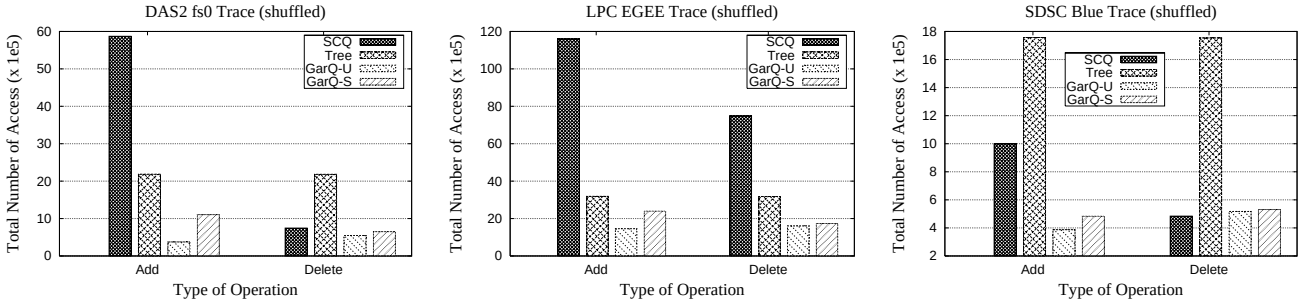

Figure 10. Total number of nodes accessed during *add* and *delete* operation using shuffled traces (shorter bars are better).

5.2 Experimental Results

5.2.1 Adding and Deleting Reservations

Figure 9 and 10 shows the total number of nodes accessed when inserting and removing reservations using the original and the shuffled traces respectively. Note that the results for List in Figure 10 are much greater than the rest, hence they are being omitted. As expected, GarQ-U and GarQ-S perform much better than other structures for the *add* operation. The only exception is when adding large jobs sequentially, as shown in Figure 9 for the SDSC Blue trace. In this case, SCQ and List perform better than GarQ-U and GarQ-S because new requests are coming in a sequential order. In addition, they are mostly being appended to the end. However, when it comes to adding new reservations when the starting time is randomized, GarQ-U performs much better overall.

Theoretically, when it comes to deleting existing reservations, SCQ with the $O(1)$ time complexity should have the best performance overall, because it only deletes the nodes in a particular array bucket. However, for the randomized traces, the performance of GarQ-U and GarQ-S for the *delete* operation is shown to be on par with SCQ. Note that SCQ performs badly on the shuffled LPC trace compare to other structures because the incoming reservations are sorted based on their starting time. Unfortunately, some reservations located in front of the list have a longer duration. Hence, in the worst case, SCQ needs to iterate through the list to delete reservations with shorter duration time.

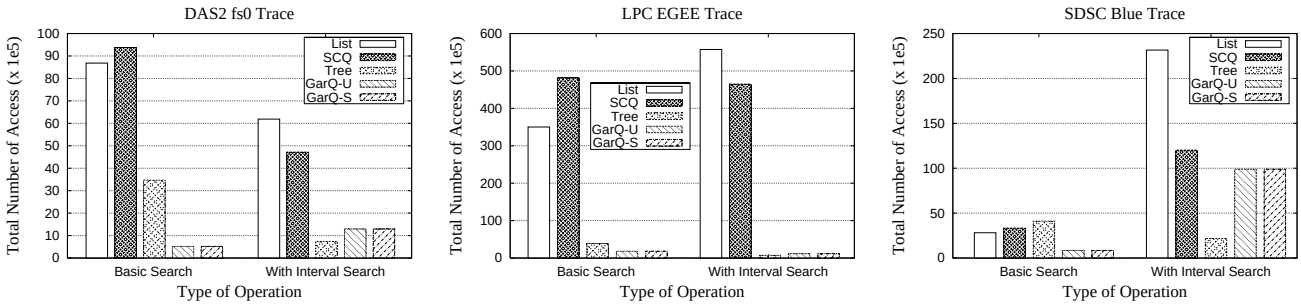

Figure 11. Total number of nodes accessed during *search* operations using original traces (shorter bars are better).

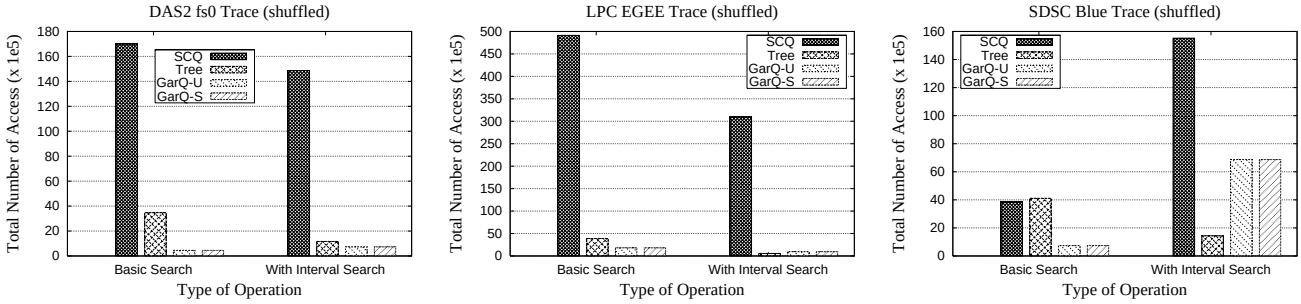

Figure 12. Total number of nodes accessed during *search* operations using shuffled traces (shorter bars are better).

5.2.2 Searching for Available Slots

Figures 11 and 12 shows the total number of nodes accessed when searching for empty slots using the original and the shuffled traces respectively. Note that for the *interval search* operation, we set the maximum time limit or MAX_LIMIT to be 12 hours from the request's initial starting time. In addition, the results for List in Figure 12 are much greater than the rest, hence they are being omitted.

These figures show that List performs the worst of all. This is because for searching, List has to start from the beginning and iterate through the effected nodes to find out the resource availability. Worse performance also incurred by SCQ compare to Tree, GarQ-U and GarQ-S, because it applies the same principle as List.

When dealing with the *search* operation, GarQ-U and GarQ-S perform the best overall, because they perform a sequential comparison. However, Tree has an advantage in the *interval search* operation, since it can find out the CNs' availability at a larger time interval. This scenario is clearly shown for the SDSC Blue trace of Figure 11 and 12.

5.2.3 Average Runtime Performance

To measure the average runtime performance of each data structure, we run the experiments several times on a 2 Ghz Opteron machine with 4 GB of RAM. We take into account the time required to perform

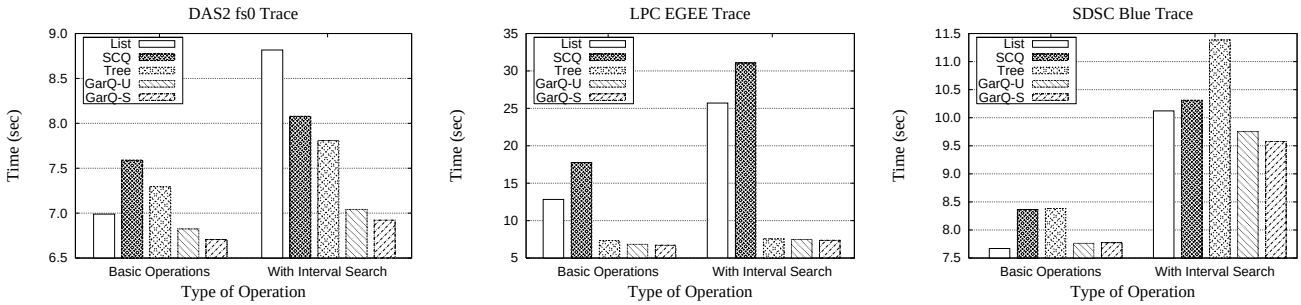

Figure 13. Average runtime using original traces (shorter bars are better).

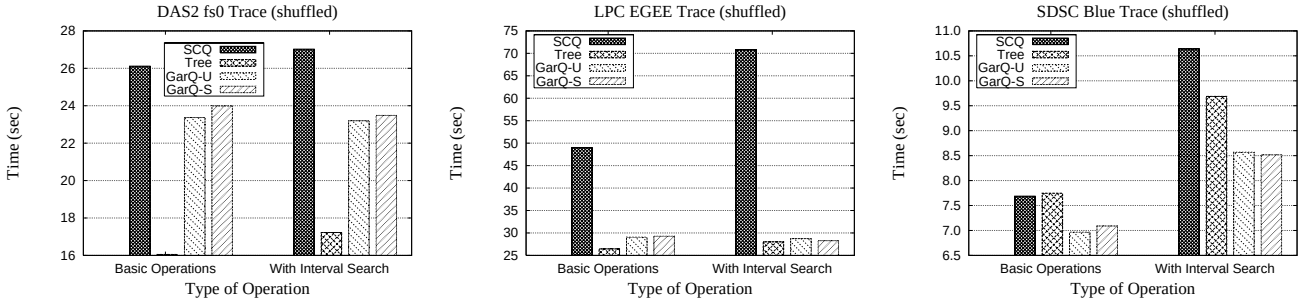

Figure 14. Average runtime using shuffled traces (shorter bars are better).

"basic operations", i.e. conducting the *add, delete* and *search* operation as a whole, and to run these operations with the *interval search* one. Figure 13 and 14 show the average runtime using the original and the shuffled traces respectively. Note that the results for List in Figure 14 are much greater than the rest, hence they are being omitted.

From Figure 13 shows that GarQ-U and GarQ-S perform the best overall. For the basic operations, SCQ performs the worst since the δ value of 60 minutes is not optimal for executing small and medium jobs of DAS2 and LPC trace respectively. For operations that include the *interval search* one, List and SCQ perform badly than the rest as expected. However, Tree took the longest time for running large jobs of the SDSC trace, partly due to the overhead of using recursive functions.

From Figure 14, for the randomized DAS2 trace, GarQ-U and GarQ-S do not perform too well compare to Tree because this trace contains many small jobs. SCQ is also take a big performance hit for these jobs. An improvement to GarQ can be done by imposing a minimum duration limit by the resource and/or grouping small jobs as one big batch before requesting a reservation. With this approach, GarQ will be able to perform more efficiently since this scenario will be similar to reserve large jobs, as shown in the SDSC Blue trace of Figure 14. On another note, the overhead cost of using the interval search operation in GarQ-U and GarQ-S is minimal compare to others. This is a very encouraging result since the array-based implementation is also easy to implement.

5.2.4 Average Memory Consumption

For measuring the average memory consumption of each data structure, we run the experiments on the same testbed as previously mentioned, on a 2 Ghz Opteron machine with 4 GB of RAM. We measure the memory consumption based on the measurement before and after the experiment. Moreover, in order to

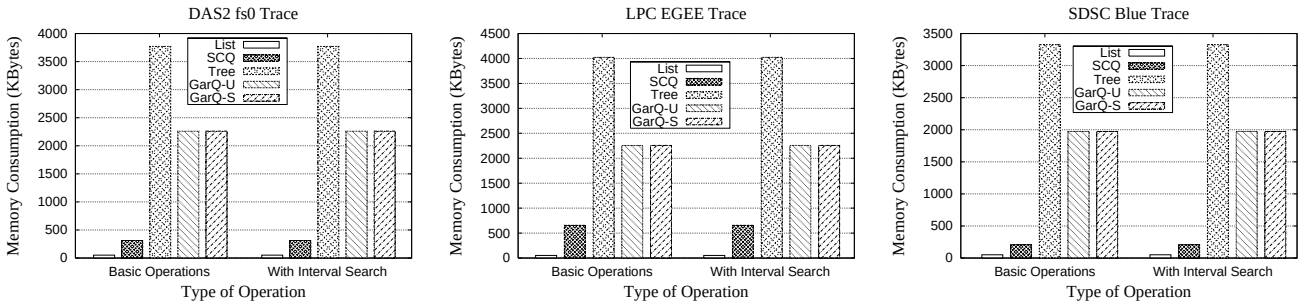

Figure 15. Average memory consumption using original traces (shorter bars are better).

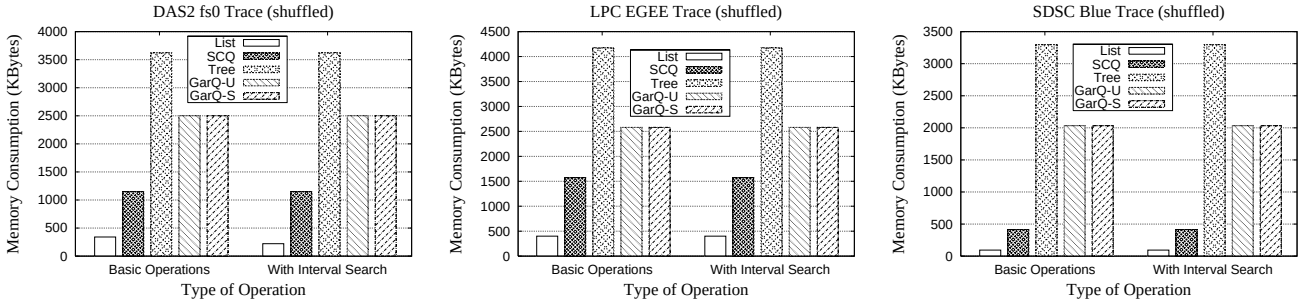

Figure 16. Average memory consumption using shuffled traces (shorter bars are better).

improve accuracy, we run the experiment several times.

From Figure 15 and 16, List and SCQ are very efficient in all of the traces, followed by GarQ-U and GarQ-S. However, SCQ requires more memory than List due to the cost of having fixed M buckets and duplicating reservations that take longer than δ across several buckets. Tree consumed more memory because the complete tree needs to be built for the whole interval length. Note that in these experiments, all data structures require less than 5 KB of RAM in a machine with a total memory of 4 GB. Therefore, the trade-off between space and time complexity can be neglected.

There is a big trade-off between low memory consumption and runtime performance. Even though both List and SCQ consume the least amount of memory, their runtime performance were the worst, as mentioned previously. Overall, GarQ-U and GarQ-S have a better ratio in all the traces.

6 Conclusion and Further Work

Advance Reservation (AR) in Grid computing is an important research area as it allows users to gain concurrent access to resources by allowing their applications to be executed in parallel. It also provides guarantees on the availability of resources at the specified times in the future. An efficient data structure is significant in minimizing the time complexity needed to perform AR operations. In this paper, we have presented a new data structure, we named it GarQ, to efficiently search for available compute nodes, to add new requests, and to delete existing reservations. We have also introduced a new operation called *interval search* for finding a free interval closest to the requested reservation, if it was previously rejected. This operation is important because it helps users in negotiating a suitable reservation time.

GarQ is an array-based data structure inspired by Calendar Queue and Segment Tree. According to

the performance evaluation, whose input is taken from real workload traces such as DAS2 fs0 from Vrije University in Amsterdam, GarQ manages to perform much better on average than Linked List, Segment Tree and Calendar Queue for the above reservation operations. However, for small jobs in the randomized DAS2 fs0 trace, Segment Tree proves to have a better average runtime performance of all. We shuffled or randomized the starting time of jobs from these traces because they are logged in increasing order. Overall, GarQ has a better ratio between low memory consumption and runtime performance compare to these data structures. Hence, the results are encouraging because our data structure is also easy to implement and can be reused for the next time interval. Therefore, it only needs to be built once in the beginning.

An extension to this work is to consider imposing a minimum duration limit by a resource and/or grouping small jobs as one big batch before requesting a reservation. With this approach, GarQ will be able to perform more efficiently since this scenario will be similar to reserve large jobs. Moreover, we are thinking of comparing the performance and effectiveness of GarQ and other data structures when dealing with the affects of scheduling issues, such as deadline, backfilling and job preemption. Finally, different types of traces and applications need to be considered in the performance evaluation.

Acknowledgement
This work is partially supported by research grants from the Australian Research Council (ARC) and Australian Department of Education, Science and Training (DEST). We would like to thank Hussein Gibbins and Srikumar Venugopal for their comments on the paper.

References

[1] D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with Nimrod/G: Killer application for the global grid? In *Proc. of the 14th International Symposium on Parallel and Distributed Processing (IPDPS)*, Cancun, Mexico, May 1–5 2000.

[2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[3] A. Brodnik and A. Nilsson. A static data structure for discrete advance bandwidth reservations on the internet. In *Proc. of Swedish National Computer Networking Workshop (SNCNW)*, Stockholm, Sweden, September 2003.

[4] R. Brown. Calendar queues: A fast $O(1)$ priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[5] L.-O. Burchard. Analysis of data structures for admission control of advance reservation requests. *IEEE Transactions on Knowledge and Data Engineering*, 17(3), 2005.

[6] R. Buyya, D. Abramson, and J. Giddy. Nimrod-G: An architecture for a resource management and scheduling system in a global computational grid. In *Proc. of the 4th Intl. Conference & Exhibition on High Performance Computing in Asia-Pacific Region (HPC Asia)*, Beijing, China, May 2000.

[7] R. Buyya and M. Murshed. GridSim: A toolkit for the modeling and simulation of distributed management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience (CCPE)*, 14:13–15, 2002.

[8] W. Cirne, F. Brasileiro, J. Sauve, N. Andrade, D. Paranhos, E. Santos-Neto, and R. Medeiros. Grid computing for bag of tasks applications. In *Proc. of the 3rd IFIP Conference on E-Commerce, E-Business and E-Government*, Sep 2003.

[9] K. B. Erickson, R. E. Ladner, and A. Lamarca. Optimizing static calendar queues. *ACM Trans. on Modeling and Computer Simulation*, 10(3):179–214, 2000.

[10] D. Feitelson. Parallel workloads archive, 2007.

[11] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[12] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Proc. of the 7th International Workshop on Quality of Service*, London, UK, 1999.

[13] J. MacLaren, editor. *Advance Reservations: State of the Art (draft)*. GWD-I, Global Grid Forum (GGF), June 2003. http://www.ggf.org.

[14] S. McGough, L. Young, A. Afzal, S. Newhouse, and J. Darlington. Workflow enactment in ICENI. *UK e-Science All Hands Meeting*, pages 894–900, September 2004.

[15] A. Oram, editor. *Peer-to-peer: Harnessing the Power of Disruptive Technologies*. O'Reilly Press, 2001.

[16] W. Smith, I. Foster, and V. Taylor. Scheduling with advanced reservations. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS)*, Cancun, Mexico, May 1–5 2000.

[17] A. Sulistio and R. Buyya. A time optimization algorithm for scheduling bag-of-task applications in auction-based proportional share systems. In *Proc. of the 17th International Symposium on Computer Architecture on High Performance Computing (SBAC-PAD)*, pages 235–242, Rio de Janeiro, Brazil, October 24–27 2005.

[18] A. Sulistio, G. Poduval, R. Buyya, and C.-K. Tham. On incorporating differentiated levels of network service into GridSim. *Future Generation Computer Systems*, 23(4):606–615, May 2007.

[19] A. Sulistio, W. Schiffmann, and R. Buyya. Advanced reservation-based scheduling of task graphs on clusters. In *Proc. of the 13th International Conference on High Performance Computing (HiPC)*, Bangalore, India, December 18–21 2006.

[20] S. Venugopal, R. Buyya, and L. Winton. A grid service broker for scheduling e-science applications on global data grids: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(6):685–699, 2006.

[21] T. Wang and J. Chen. Bandwidth tree – a data structure for routing in networks with advanced reservations. In *Proc. of the 21st Intl. Performance, Computing, and Communications Conference (IPCCC)*, pages 37–44, Phoenix, USA, 2002.

[22] Q. Xiong, C. Wu, J. Xing, L. Wu, and H. Zhang. A linked-list data structure for advance reservation admission control. In *Proc. of the 3rd International Conference on Networking and Mobile Computing (ICCNMC)*, Zhangjiajie, China, August 2-4 2005.

[23] L. Yuan, C.-K. Tham, and A. L. Ananda. A probing approach for effective distributed resource reservation. In *Proc. of the 2nd International Workshop on Quality of Service in Multiservice IP Networks*, pages 672–688, Milan, Italy, February 2003. Springer-Verlag.