# Dynamic Decentralized Preemptive Scheduling across Heterogenous Multi-core Desktop Grids

Arun Balasubramanian
Department of Computer Science
University of Maryland College Park
arunb@cs.umd.edu

## Abstract

*The recent advent of multi-core computing environments increases the heterogeneity of grid resources and the complexity of managing them, making efficient load balancing challenging. In an environment where jobs are submitted regularly into a grid which is already executing several jobs, it becomes important to provide low job turn-around times and high throughput for the users. Typically, the grids employ a First Come First Serve (FCFS) method of executing the jobs in the queue which results in poor average job turnaround times and wait times for most jobs. Hence a conventional FCFS (First come first serve) scheduling strategy does not suffice to reduce the average wait times across all jobs.*

*In this paper, we propose new decentralized preemptive scheduling strategies that backfill jobs locally and dynamically migrate waiting jobs across nodes to leverage residual resources, while guaranteeing bounded turn-around and waiting times for all jobs. The methods attempt to maximize total throughput while balancing load across available grid resources. Experimental results for intra-node scheduling via simlulation show that our scheduling scheme performs considerably better than the conventional FCFS approach of a distributed or a centralized scheduler.*

## 1. Introduction

Modern desktop machines use multi-core CPUs to enable improved performance. Contention or shared resources can make it hard to exploit multiple computing resources efficiently and so, achieving high performance on multi-core machines without optimized software support is still difficult[1]. Moreover, desktop grids that contain multi-core machines are becoming increasingly diverse and heterogeneous[4], so that efficient load balancing and scheduling for the overall system is becoming a very challenging problem[6][7] even with global status information and a centralized scheduler[3].

Previous research[17] on decentralized dynamic scheduling improves the performance of distributed scheduling by starting waiting jobs capable of running immediately (backfilling), through use of residual resources on other nodes (if the job is moved) or on the same node (if the local schedule is changed). However, the scheduling strategy is non-preemptive and follows a First come first serve approach to schedule the jobs. This results in poor average wait times and turn around times for jobs in the queue. This also results in poor overall Job Throughput rate in the grid.

The performance of distributed scheduling and overall job throughput of distributed scheduling in such multicore environments can be improved by following a preemptive scheduling strategy where jobs that wait longer in the queue get a chance to run. The techniques of migrating jobs to use residual resources on neighboring nodes can also be used to increase the overall CPU utilization. However because of limited and/or stale global state, efficient decentralized job migration can be difficult to achieve. Moreover, a job profile often has multiple resource requirements; a simple job migration mechanism considering only CPU usage cannot be applied to in such situations. In addition, guarantee of progress for all jobs is also desired, i.e., no job starvation.

The Contribution of this paper is a novel dynamic preemptive scheduling scheme for multi-core desktop grids. The scheme includes (1) local preemptive scheduling, with backfilling on a single node, (2) internode scheduling, for backfilling across multiple nodes, and (3) queue balancing, which proactively balances wait queue lengths. The approach inspires ideas from the preemptive schedulers in the context of operating systems and schedules jobs at regular intervals based on its priorities. The priorities of the jobs are determined according to their remaining time for completion and the amount of time the job has spent waiting in the queue. It is a completely decentral-

ized scheme that balances load and improves throughput when scheduling jobs with multiple constraints across a distributed system. We demonstrate the effectiveness of our algorithms via simulations that show that the decentralized approach performs competitively with an online centralized scheduler.

The rest of this paper is organized as follows. Section 2 discusses the Related Work on various preemptive strategies in literature. Section 3 discusses and describes the basic architecture of the peer-to-peer desktop grid system and the resource management schemes for multi-core machines. The term definitions related to the scheduling algorithm are presented in section 4. The Preemptive scheduling approach is discussed in Section 5. The simulation results are presented in Section 6. Conclusion and Future work are presented in Section 7 and Section 8 respectively.

## 2. Related Work

Various preemptive scheduling algorithms exist in literature in the contexts of Operating Systems, Batch Processing environments and Real time scheduling. Round Robin[11] is the initial and simplest algorithm for a preemptive scheduler where only a single queue of processes is used. When the system timer fires, the next process in the queue is switched to, and the preempted process is put back into the queue. Classical UNIX systems [12] [13] scheduled equal-priority processes in a round-robin manner, each running for a fixed time quantum. If a higher priority process becomes runnable, it will preempt the current process (if it's not running in kernel mode, since classical UNIX kernels were non-preemptive) even if the process did not finish its time quantum. This way, high priority processes can possibly starve low-priority ones. To avoid this, a new factor in calculating a process priority was introduced: the 'usage' factor. This factor allows the kernel to vary processes priorities dynamically. When a process is not running, the kernel periodically increases its priority. When a process receives some CPU time, the kernel reduces its priority. This scheme will potentially prevent the starvation of any process, since eventually the priority of any waiting process will rise high enough to be scheduled. While the Operating system schedulers usually act on the basis of information obtained from the processes execution so far and the priority of processes, the Batch processing and real time schedulers have added information like estimated job completion times and deadlines respectively.

Our environment closely resembles that of the Batch Processing scenario since it is reasonable to obtain estimates on the job completion times. Shortest remaining time[14] is a scheduling method that is a preemptive version of shortest job next [15] scheduling. In this scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time. This leads to higher wait times for long running jobs. Highest Response Ratio Next (HRRN)[16] scheduling is a non-preemptive discipline, in which the priority of each job is dependent on its estimated run time, and also the amount of time it has spent waiting. Jobs gain higher priority the longer they wait, which prevents indefinite postponement (process starvation). i.e. the jobs that have spent a long time waiting compete against those estimated to have short run times. In this paper, we use the idea of 'Higher Response Ratio Next' in a preemptive environment to ensure that long running jobs are not starved of CPU usage while at the same time guaranteeing that shorter jobs finish early. This contributres to the overall high throughput in the system.

## 3. BACKGROUND

(Content in this section is based on [17])

### 3.1. Overall System Architecture

Prior to this, a completely decentralized peer-to-peer (P2P) desktop grid system has been developed that is both resilient to single-point failures, and provides good scalability[8]. A desktop grid system exists with heterogeneous nodes with different resource types and capabilities, e.g. CPU speed, memory size, disk space and number of cores. Jobs submitted to the grid also can have multiple resource requirements, limiting the set of nodes on which they can be run. It is assumed that every job is independent, meaning that there is no communication between jobs. To build the P2P grid system, a variant of a Content Addressable Network (CAN) [9] distributed hash table (DHT) is employed, which represents a nodes resource capabilities (and a jobs resource requirements) as coordinates in a d-dimensional space. Each dimension of the CAN represents the amount of that resource, so that nodes can be sorted according to the values for each resource. A node occupies a hyper-rectangular zone that does not overlap with any other nodes zone, and the zone contains the nodes coordinates within the d-dimensional space. Nodes exchange load and other information with nodes whose zones abut its own (called neighbors). The following steps describe how jobs are submitted and executed in the grid system.

1) A client(user) inserts a job into the system through an arbitrary node called the injection node.

2) The injection node initiates CAN routing of the job to the owner node.
3) The owner node initiates the process to find a lightly loaded node (runnode) that meets all of the jobs resource requirements (called matchmaking)
4) The run node inserts the job into an internal FIFO queue for job execution. Periodic heartbeat messages between the run node and the owner node ensure that both are still alive. Missing multiple consecutive heartbeats invoke a (distributed) failure recovery procedure.
5)After the job completes, the run node delivers the results to the client and informs the owner node that the job has completed.
The owner node monitors a jobs execution status until the job finishes and the result is delivered to the client. To enable failure recovery, the owner node and the runnode periodically exchange soft-state heartbeat messages to detect node failures (or a graceful exit from the system). More details about the basic system architecture can be found in Kim et al. [9].

### 3.2. Matchmaking Procedure

Matchmaking is the initial job assignment to a node that satisfies all the resource requirements of the job, and also does load balancing to find a (relatively) lightly loaded node. A good matchmaking algorithm has several desirable properties: expressiveness, load balance, parsimony, com-pleteness, and low overhead. The matchmaking framework should be expressive enough to specify the essential resource requirements of the job as well as the capabilities of the nodes. It should balance load across nodes to maximize total throughput and to obtain the lowest job turnaround time. However, over-provisioning can decrease total system throughput, therefore the matchmaking should be parsimonious so as not to waste resources. Completeness means that as long as the system contains a node that satisfies a jobs requirements, the matchmaker should find that node to run the job. Finally, the overall matchmaking process should not incur signicant costs, to minimize overhead.
The CAN-based decentralized matchmaking framework directly supports expressiveness and completeness with low overhead. The previous efforts to enhance load balancing performance but be parsimonious are two-fold-employing a virtual dimension and using probabilistic pushing of jobs. The basic CAN mechanisms do not allow the multiple nodes to have the same coordinates in the multidimensional space.However, the coordinates in our CAN are determined by the amount of each resource a node has, so multiple nodes with identical resource capabilities can conflict. This problem is addressed by adding another dimension (called the virtual dimension), which has a random value assigned to differentiate multiple nodes with the same capabilities. The random value in the virtual dimension also helps distribute jobs across nodes evenly, so improves load balance. However, using the virtual dimension does not always achieve good load balance.
The basic matchmaking algorithm has been modified to improve load balance by pushing jobs into less loaded regions in the CAN in a probabilistic way. The global load information is aggregated along each dimension by piggybacking load data onto the periodic heart beat messages sent between neighbors that are used to maintain the CAN structure. After a job is routed to the node that meets its minimum resource requirements, that node chooses a dimension and a target node among its neighbors, to try to find a path to a more lightly loaded region in the CAN. The decision process to push the job employs the periodically updated aggregate load information along each dimension. However, before pushing the job, the node computes a stopping probability based on known load information in outer regions of the CAN, to determine whether the job is to be pushed or not. If a job stops at a node, that node will pick as the run node the least loaded node among itself and its neighbors. Otherwise,the job continues to be pushed to a node with higher resource capability farther out in some dimension in the CAN. This probabilistic approach can balance load effectively, but also minimizes over-provisioning. More details on this work for initial job placement can be found in Kim et al. [10].

### 3.3. Resource Management in a Multi-core Grid

Multi-core nodes may be capable of running multiple jobs simultaneously, so that the number of currently available cores and the available amount of other shared resources can vary over time for each node in the grid. Jobs also may request more than one core to express the requirements of a multi-threaded application. However, a structured DHT like CAN can have problems with frequent changes to its structure, because it works best in a low-churn environment. To express the dynamically changing amount of available resources in each node,and to minimize the changes required to the existing CAN mechanisms, a dynamic resource availability is represented by employing two logical nodes for each physical one: one that models the maximum resource available for that node (Max-node),and a second that models the currently unused amount of that resource(Residue-node)[2].
Two resource management schemes have been designed, named Balloon-Model and Dual-CAN, that employ two logical nodes per physical node. Dual-CAN uses two separate CANs, one for each logical node type, so that dynamic effects due to resource changes (e.g.,jobs starting or end-

ing) in a multi-core node affect only the Secondary CAN, which contains only Residue-nodes. The number of nodes in the Secondary CAN is typically much fewer than in the Primary CAN (composed of Max-nodes), so the additional overhead for managing the Dual-CAN is not high. However, maintaining an additional CAN is not free, so its also possible to incorporate Residue-nodes into a single Primary CAN in a simple form, called a Balloon. A Balloon represents the currently available amount of resources for a node as a point in the CAN, and is associated with the zone that contains that point in the CAN. Therefore, the addition or removal of a Balloon due to resource availability changes for the node the Balloon represents affects atmost 2 nodes in the CAN, minimizing changes to the Primary CAN. Using both static and dynamic node information in the two management schemes, a job is assigned to an appropriate node capable of running the job, preferably a node not currently running any other jobs (a free node). The initial job matchmaking and information aggregation schemes are similar to what was described for a single-core environment in Section 3.2, except that the algorithms require information on core utilization rather than on the number of free nodes. Once a run node is determined, the job is inserted into the local queue of the node to wait to be run. The default queuing policy is first-come first-serve (FCFS), based on the time the job arrived in the system, but a node tries to run as many jobs as possible simultaneously to utilize all its available resources.

## 4. Term Representations

1) An arbitrary job in queue (Non executing) job = $J_a$.
2) Currently Running (or executing) Job = $J'_a$
3) Priority of Job $J_a$ = $P_{j_a}$
4) Priority of currently running job = $P_{j'_a}$
5) Job at head of queue = $J_h$
6) Priority of Job at head of queue $J_H$ = $P_{j_h}$
7) Remaining Time for Job $J_a$ = $T_{rem}(J_a)$
8) Resource requirements for Job $J_a$ = $R_{j_a}$
9) Current free residual resources = $R_f$
10) Residual resources that would be available when some current running jobs are preempted = $R_f(temp)$
10) Resource requirements of Job at head of queue = $R_{j_h}$
11) Resource requirements of Job currently running = $R_{j'_i}$
12) Minimum Priority Job running currently in a given set = $J'_{Pmin}$
13) Minimum Resource consuming Job running currently in a given set= $J'_{Rmin}$
14) Resource requirements of $J'_{Pmin}$ = $R_{j'_{Pmin}}$
15) Resource requirements of $J'_{Rmin}$ = $R_{j'_{Rmin}}$
16) Waiting time of Job $J_a$ = $W_{J_a}$
17) Jobs covered so far for analysis = $J_{covered}$
18) Jobs currently running = $J_{running}$
19) The remaining jobs (those yet to be examined for preemption) = $J_{rem}$
20) Priority of the Highest priority job that is covered so far = $P_{max}(J_{covered})$
21) Priority of the Lowest priority job that is covered so far = $P_{min}(J_{covered})$
22) Resources required for the Minimum Resource consuming job that is covered so far = $R_{min}(J_{covered})$
23) Resources required for the Maximum resource consuming job that is covered so far = $R_{max}(J_{covered})$
24) Priority of the Highest priority job from the remaining jobs (those yet to be examined for preemption) = $P_{max}(J_{rem})$
25) Priority of the Lowest priority job from the remaining jobs (those yet to be examined for preemption) = $P_{min}(J_{rem})$
26) Resources required for the Minimum Resource consuming job from the remaining jobs (those yet to be examined for preemption) = $R_{min}(J_{rem})$
27) Resources required for the Maximum resource consuming job from the remaining jobs (those yet to be examined for preemption) = $R_{max}(J_{rem})$

## 5. PREEMPTIVE SCHEDULING

### 5.1. Local Scheduling

This section deals with the scheduling criteria for a single node. As mentioned in section 2, we combine the ideas of 'shortest remaining time next' and the 'higher response ratio next' to come up with a preemptive scheduling algorithm for the desktop grids. The 'shortest remaining time next' ensures that jobs that have the smaller remaining time are run, so they end sooner. However, this could lead to starvation for long running jobs and hence we increase the priority for jobs that wait longer in the queue. Thus, the jobs waiting in the node's queue have their priorities calculated as

$$P_{j_a} = 1 + (\alpha * W_{J_a})/T_{rem}(J_a)$$

i.e. the priority for a job is directly proportional to its wait time $W_{J_a}$ and inversely proportional to its estimated time for completion $T_{rem}(J_a)$. $\alpha$ is the weight factor associated with the wait time. Typically, the $\alpha$ value is greater than 1. The section on 'Experimental Results' provide more details on the values of $\alpha$.
The job queue is sorted according to the order of their priorities calculated as above. Initially, the jobs in the head of the queue are scheduled until the available resources are insufficient for the next job to run. Next, those jobs that can run in the available residual resources are scheduled to run (Backfilling). Since the backfilled jobs have priorities

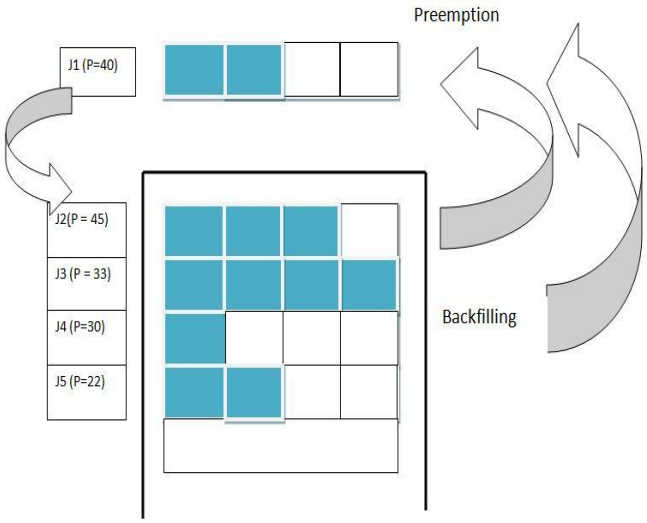


Figure 1. Local Preemptive scheduling

associated with them, they are also prone to preemption and therefore do not starve jobs waiting in the queue.

The scheduler is invoked at the following 3 phases in the system:
1) After, every periodic scheduling interval $\delta$.
2) As a new job enters the queue.
3) A job completes its execution.
The periodic interval $\delta$ is much higher when compared to the scheduling intervals for schedulers in the OS. This is because a context switch
between jobs is more expensive (in terms of time) when compared to context switch between processes. And so, frequent context switches would result in low overall CPU utilization. More details regarding the values of $\delta$ are discussed in the Results section.
The scheduler is also invoked when a new job enters the queue because the newly arrived job could be backfilled. And, when a job completes execution, it frees up some resources which allows new jobs to run. At every scheduling turn, the priorities of job in the head of queue is compared with that of the least priority job thats currently running. This is done because the job at the head of the queue cannot run currently if its priority is lower than the lowest priority job thats currently running. This also addresses the backfilled jobs immediately since backfilled jobs have the lowest priority among the running jobs.
If the priority of the job at head of queue($P_{j_h}$) is greater, the scheduler checks if the current running job $J'_{Pmin}$ frees up enough resources for the new job to run. If yes, the job $J'_{Pmin}$ is preempted and $J_h$ is scheduled. Otherwise, the scheduler compares the prioritiy of the second lowest priority job ($J'_a$) with $J_h$. This is carried out until the scheduler appropriately preempts jobs that free up just the right amount of resources for the job $J_h$ to run. If the scheduler is unable to free up sufficient resources for the job to run, the job $J_h$ is not scheduled in this interval and has to wait until the next scheduling turn. The scheduling (at every scheduling turn) is carried out for all jobs in the queue that have a higher priority than the lowest priority job that is currently running. More details are presented in the algorithm.

### 5.2. Algorithm for Scheduling

1. Job queue is sorted based on Job Priority.

2. Calculate priority for each job as :
a. $P_{j_a} = 1 + ((\alpha * W_{J_a})/T_{rem}(J_a))$

3. Schedule all jobs from the head of queue until the available resources run out for a job.

4. Then, look for other jobs that can run in residual resource. Schedule them if a match is found.
While $(R_f! = null)$
{
i.e. Find $J_a$ such that $R_{j_a} < R_f$. If found such a $J_a$,
$J_a -> J'_a$
$R_f = R_f - R_{j_a}$
}
5. In the next Scheduling turn, compare the priorities of job in the head of the Q with that of the least priority job that is currently running.
$J_h$ = next job in queue
While $((P_{j_h} > P_{min}(J_{running})||P_{j_h}! = null)$
{
$J_{covered} = null$;
$J_{rem} = null$;
$R_f(temp) = R_f$
$J_{covered} = J'_{Pmin}$
if $R_{j_h} <= (R_{j'_{Pmin}} + R_f(temp))$
Preempt $J'_{Pmin}$ and schedule $J_h$
else {
$R_f(temp) = R_f(temp) + R_{j'_{Pmin}}$ and Goto step 6.
}

6. While $(J_{rem}! = null)$
{
Choose new job $J'_a$ such that $P_{j'_a} > P_{max}(J_{covered})$ and $P_{j'_a} = P_{min}(J_{rem})$
$J_{covered} += J'_a$
$J_{rem} = J_{running} - J_{covered}$
If $P_{j'_a} > P_{j'_h}$
{
cannot preempt jobs;

Wait for next scheduling turn;
break;
}
else
{
If $R_{j_h} >= R_{j'_a}$
then, only preempt $J'_a$ and schedule $J_h$
break;
else
Goto Step 7
}

7. if $R_{j'_a} >= (R_{j_h} - R_f)$
(
From among the list of $J'_i$ with $P_{j'_i} < P_{j'_a}$ choose $J'_i$ such that
$R_{j'_i} = R_{J'_{Rmin}}$ and check
if $R_{j'_a} + R_{j'_i} >= R_{j_h}$ then Preempt $J'_i$ ,$J'_a$ and schedule $J_h$.
else
continue searching for $J'_i$ incrementally with respect to the resource requirements.
break;
)
else
$R_f(temp) = R_f(temp) + R_{j'_i}$
Proceed to step 6.
}
$J_h$ = next job in queue;
}

### 5.3. Internode Scheduling

Internode scheduling is an extended version of local scheduling; the target node for backfilling can be the neighbors in the CAN. Local scheduling only deals with changes to the job execution order within the queue on a node. Internode scheduling however, must decide the following:
1) Which node initiates job migration,
2) Which node should be the sender of a job,
3) and which job should be migrated.
Internode scheduling takes place periodically at every scheduling interval after the local scheduling process to see if the job at the top of the queue in the node can be run on any of its neighbors and also to see if the node can run the job of any of its neighbors in its currently free residual resources.
In the PUSH scheduling model the job sender initiates the migration process. First the sender node tries to match priority of the job at the head of the queue with the neighboring nodes queue. If the priority of the job at head of the queue in its neighbor node is less than the job at sender

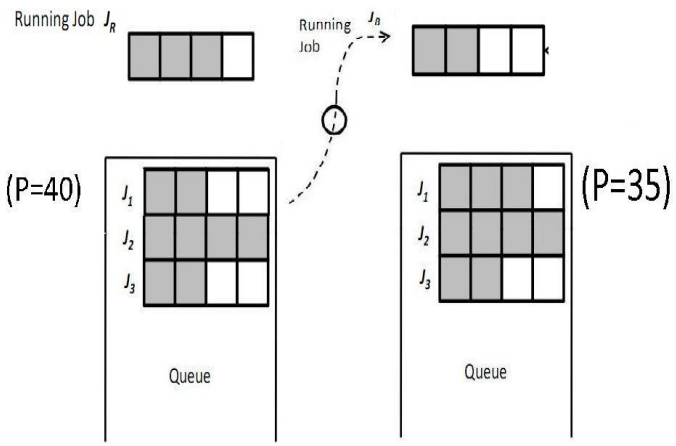


Figure 2. Internode scheduling

node, a PUSH message for the job is sent to its neighbor containing the priority and the resource requirements. If the job can be backfilled at the neighbor node, the PUSH message is accepted. Otherwise, a PUSH-reject message is sent back to the sender node. If a job can be run on multiple neighbors, the sender sends it to the node that has minimum objective function value as follows.

$$f_{Inter-PUSH} = BM * FM * (1/CPU_{speed})$$

To prefer the fastest node among neighbors, the objective function also includes an inverse term for CPU speed. Before sending a job profile, there is a simple confirming handshake process between a sender and a potential receiver to avoid inappropriate job migration because the potential receiver information may not be up-to-date at the sender.

In the PULL model, a receiver node tries to obtain a job from its CAN neighbors so as not to waste its available resources. However, the node does not have all information on the queued jobs resource requirements in its neighbors to minimize neighbor update message sizes, so the node invokes a PULL-Request message to the node having the closest priority job at the head of queue that is higher than the priority of job at the head of the queue in the current node. If there are multiple such nodes, the request is sent to the node with maximum queue size among its neighbors. If there are multiple candidate jobs in the waiting queue, then the job that has minimum objective function value (BM * FM, as above), is selected. If there is no candidate job, then the requesting node gets a PULL- Reject message and

continues to look for another potential sender having the appropriate priority along with maximum queue length not contacted recently.

### 5.4. Queue Balancing

Queue Balancing is a technique to proactively distribute the loads across nodes so that the job loads are equally balanced between the nodes. This ensures that the resources on all the nodes are efficiently utilized at any point in time either through local scheduling, backfilling or internode scheduling. Although the design techniques for Queue Balancing have been analysed to some extent, the details are not discussed here since the algorithm has not been implemented and the design ideas themselves may need more introspection. Hence, we discuss its details in the Section 7.

## 6. Experiment and Results

### 6.1. Experimental Setup

A synthetic workload was generated to model the grid resource configuration containing heterogenous nodes capable of executing a heterogenous set of jobs. The simulation scenario consists of 1000 multi-core nodes (having 1 , 2, 4 or 8 cores), and 5000 jobs submitted to run on those nodes. Each node has multiple resource capabilities for CAN resource dimension such as CPU speed, memory size, disk space and the number of cores. The jobs are also modeled similarly having the heterogenous resource configuration as their requirements. A high percentage of the nodes (and jobs) have relatively low resource capabilities (requirements), and a low percentage of nodes (jobs) have high resource capabilities (requirements).
The interval between job submissions follow a poisson distribution, with varying average job inter arrival times in the experiments. Each job has an estimated running time associated with it. The estimated times are uniformly distributed between 0.5T and 1.5T, with T= 3600 seconds, running on a canonical node with a normalized CPU speed of 1. The simulated job running time is then scaled up by the CPU speed relative to the canonical node.
We compare our schemes to the FCFS scheduler with backfilling which schedules jobs in the order they arrive and also performs backfilling of jobs on residual resources. To measure the performance of the long running desktop grid system, we run the simulations in a steady state environment. By steady state, its implied that the job arrival and departure rates are similar, so that the system acheives a dynamic equilibrium state during the simulation period, with the system neither highly overloaded, nor very underloaded. Hence, the average total system load is determined by the inter-job arrival rate. However very lightly loaded systems were not tested, because those are not very interesting for measuring dynamic scheduling performance.

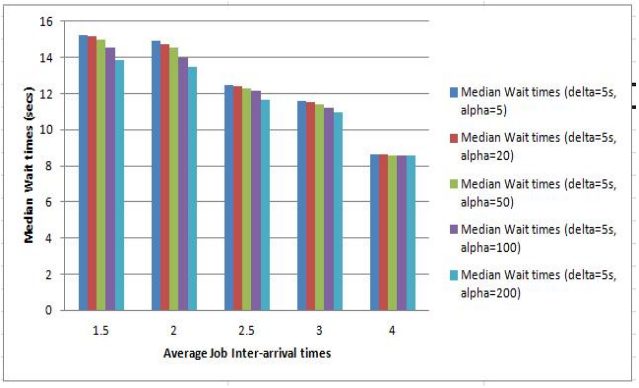

Figure 3. Median wait times for different Job inter-arrival times

### 6.2. Experimental Results

Figure 3 lists and compares the median wait times across all jobs for each job inter-arrival times. The effect of $\alpha$ (the weight associated with the wait time of a job) on the median wait times is observed for each job inter-arrival times. For lower $\alpha$ values, the job with the lesser time to complete gains more priority and as a result the current running jobs execute for longer periods without getting preempted. This results in a lower preemption frequency and hence higher wait times for jobs. As $\alpha$ increases, more priority is given to the waiting time of the job and so jobs are preempted more frequently. This results in a better performance (lower median wait times).

We also observed that the preemptive scheduling algorithm works best for smaller scheduling intervals $\delta$ i.e. they resulted in the low median and average wait times for jobs. The graphs are plotted for $\delta$ = 5 seconds and we can observe the low median wait times in all cases. As expected, the median wait times for jobs decrease as the inter arrival time for jobs increase since jobs are submitted later in the system.

In contrast however, the 'average' wait times (see figure 4) across the jobs tend to increase with increase in inter job arrival time. Although it may seem perplexing at first, this trend can be explained. As the jobs arrive later, the jobs that were initially running executed for longer periods of time. So, their estimated time for completion is now much lesser when compared to the jobs that have newly arrived. This results in continued execution of the old jobs until they complete i.e., as the inter arrival times become larger, the scenario resembles that of the FCFS approach with no pre-

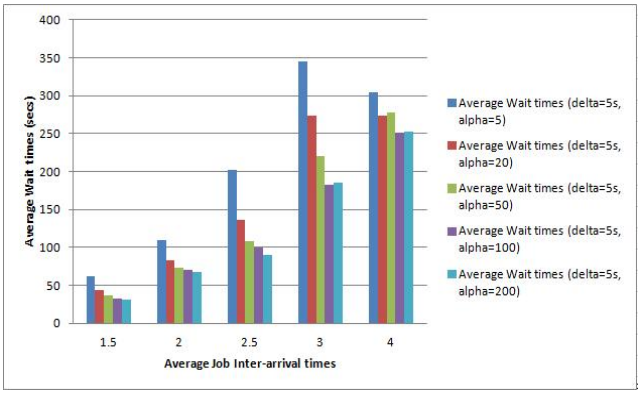


Figure 4. Average wait times for different Job inter-arrival times

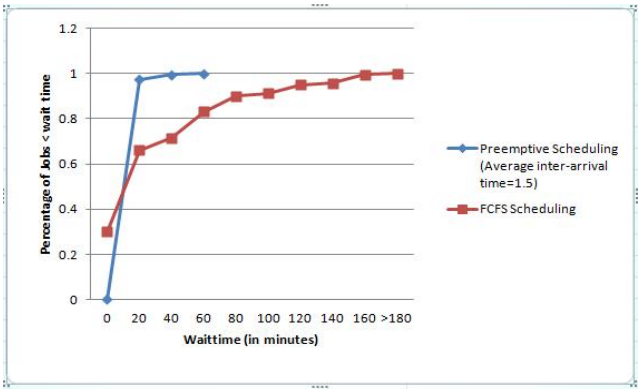


Figure 5.

emption. Hence, the newly arriving jobs have longer wait times. It can also be observed that in some cases (inter-arrival times 3 and 4) very high values of alpha (200) could result in a slight performance degradation. This is because jobs that are near completion do not finish immediately due to excessive preemption. Figures 5 to 9 illustrate the distribution of the wait times for jobs in both environments i.e. preemptive and non preemptive FCFS scheduling. As expected, the waiting times of jobs decrease with increasing job inter-arrival times in the FCFS environment. In the preemptive environment however, most jobs have the least wait times when the job inter arrival time is minimum. This is because, jobs arrive quicker and hence the earlier jobs that were executing are not very close to their completion and so are preempted by the new jobs. This results in low waiting times across all jobs. As explained earlier, when the job inter arrival times increase considerably, the preemptive scheduler behaves more as an FCFS scheduler. This can be observed in figure 9.

## 7. Conclusion

A preemptive scheduling algorithm (with backfilling) for desktop grids was designed and implemented. As part of

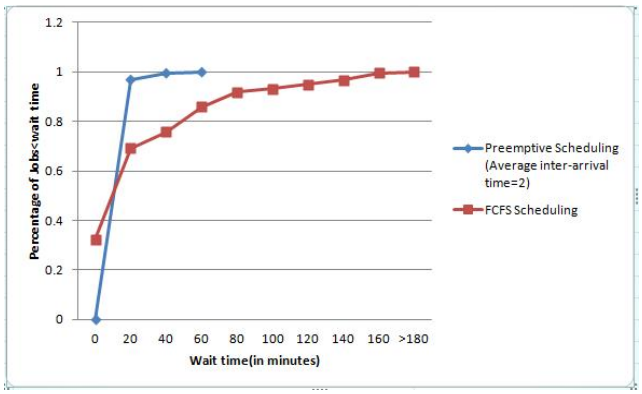


Figure 6.

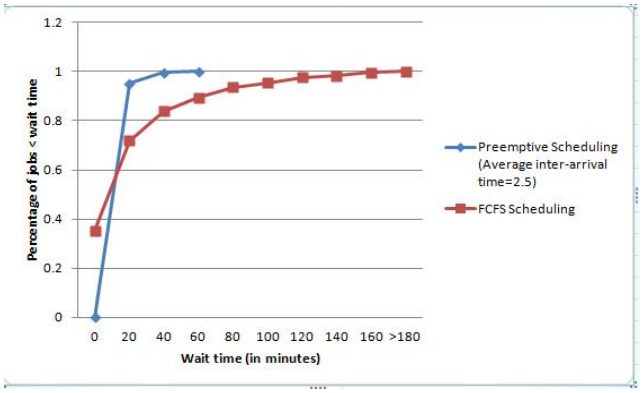


Figure 7.

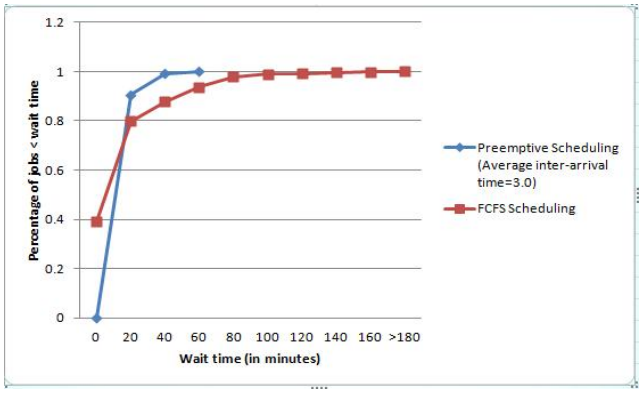


Figure 8.

local scheduling, jobs that are estimated to complete sooner were given higher priority compared to long running jobs while at the same time ensuring that the long running jobs get their fair share of the CPU. In other words, the jobs that have waited too long compete with those having short remaining times. Results show that this algorithm performs much better and yield lower average job wait times when compared to the FCFS approach. The Internode scheduling ensures that those jobs that cannot be immediately scheduled are PUSHED to a neighboring node if it can be run in

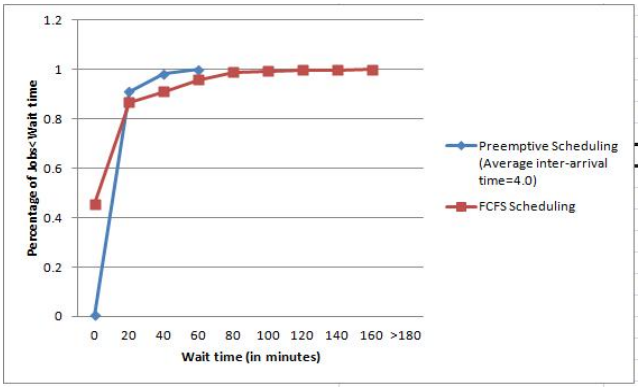


Figure 9.

the neighboring nodes residual resources. It also allows a node to PULL jobs from neighboring nodes to utilize its local residual resources. Queue Balancing assures a proactive method for balancing the load across nodes.

## 8. Future Work

The Internode scheduling algorithm has to be implemented and the results have to be studied. Since the results of the local scheduling algorithm are very promising, we expect the Internode scheduling to further bring down the waiting times for jobs and increase the average throughput.

The local scheduling and internode scheduling algorithms find and execute a job using residual free resources in a node. This means that only jobs that can start running immediately will be moved. However, if the load across nodes is skewed, the job queue lengths vary greatly, and hence a more pro-active queue balancing scheme would improve load distribution, and overall throughput, across heterogeneous nodes. The grid model allows for multiple resource types to be specified for a node, therefore defining and measuring load is more complex than for a single resource type. Firstly, the maximally loaded resource among the K available resources is set as the Load of a node, and the algorithm minimizes the total sum of the Loads among neighbors, and also balances Load across the nodes[5]. The term $W_i^k$ is defined, normalized load for Resource $k$ of Node $i$ by:

$$W_i^k = \sum_{J_j \in Queue_i} (R_j^k), 1 \leq k \leq K$$

where $J_j$ is Job j, $R_j^k$ is the kth normalized resource requirement for $J_j$, and $Queue_i$ is the job queue for node i. The normalized load of Node i, $L_i$ is given by

$$L_i = Max(W_k^i), 1 \leq k \leq K$$

The PUSH and PULL job migration models can be used for queue balancing, as they were for internode scheduling. For PUSH, a node i computes normalized load($L_i$) for itself and for its neighbors. If $L_i$ is the locally maximum value among all its neighbors, then node i checks its queue to find candidate jobs for migration that reduce $L_i$ if the (candidate) job is moved. Among these jobs, those jobs that satisfy the priority constraints in the neighboring node are considered. When there are multiple candidate jobs, the algorithm selects the job and the receiver node that minimize an objective function if the job is moved to the neighbor.

The PULL model is similar to the PUSH model, except that the node with a locally non-zero minimum normalized load among equal or less capable neighbors will initiate the PULL process from the most loaded node among its neighbors. The Queue Balancing and the Internode Scheduling techniques are expected to further improve the performance of the desktop grid system.

## 9. References

[1] S.Moore, Multicore is bad news for super computers, IEEE Spectrum, vol.45, no.11, pp.1515, Nov.2008.

[2] J.Lee, P.Keleher and A.Sussman, Decentralized resource management for multi-core desktop grids, in Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium. Atlanta, Georgia, USA: IEEEComputer SocietyPress,2010.

[3] W.Leinberger, G.Karypis, and V.Kumar, Job scheduling in the presence of multiple resource requirements, in Supercomputing99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing(CDROM). NewYork, NY, USA: ACM, 1999, p.47.

[4] Jaehwan Lee, Pete Keleher and Alan Sussman. Supporting Computing Element Heterogeneity in P2P Grids. In Proceedings of the IEEE Cluster 2011 Conference. September 2011. IEEE Computer Society Press.

[5] W.Leinberger, G.Karypis, V.Kumar, and R.Biswas, Load balancing across near-homogeneous multi-resource servers, in Proceedings of the 9th Heterogeneous Computing Workshop, 2000.(HCW2000), 2000, pp.6071, appears with the Proceedings of IPDPS 2000.

[6] D.Zhou and V.Lo, Wavegrid: a scalable fast-turnaround heterogeneous peer-based desktop grid system, in Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS2006). IEEE Computer Society Press, April2006.

[7] Wave scheduler: Scheduling for faster turnaround time in peer-based desktop grid systems, in Proceedings of the 11th Workshop on Job Scheduling Strategies for Parallel Processing, June2005.

[8] J.-S.Kim, B.Nam, P.Keleher, M.Marsh, B.Bhattacharjee, and A.Sussman, Resource Discovery Techniques in Distributed Desktop Grid Environments, in Proceedings of the 7thIEEE/ACM International Conference on Grid Computing-GRID2006, Sep.2006.

[9] S.Ratnasamy, P.Francis, M.Handley, R.Karp, and S.Shenker, A Scalable Content Addressable Network, in Proceedings of the ACMSIGCOMM Conference, Aug.2001.

[10] J.-S.Kim, P.Keleher, M.Marsh, B.Bhattacharjee, and A.Sussman, Using Content-Addressable Networks for Load Balancing in Desktop Grids, in Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing(HPDC-16), Jun.2007.

[11] M. Shreedhar and G. Varghese, Efcient Fair Queuing Using Decit Round-Robin, IEEE/ACM Transactions on Networking, vol. 4,3, pp. 375-385, 1996.

[12] Ken Thompson, UNIX Implementation, 2.3 - Synchronization and Scheduling, Bell Laboratories

[13] Maurice J. Bach, The Design of the UNIX Operating System, Chapter 8 - Process Scheduling and Time, Prentice Hall

[14] Harchol-Balter, Mor; Schroeder, Bianca; Bansal, Nikhil; Agrawal, Mukesh (2003). Size-Based Scheduling to Improve Web Performance. ACM Transactions on Computer Systems 21 (2): 207233. doi:10.1145/762483.762486

[15] William Stallings: Operating systems: internals and design principles. 4th ed., Prentice-Hall, 2001, ISBN 0-13-031999-6.

[16] Tanenbaum, A. S. (2008). Modern Operating Systems (3rd ed.). Pearson Education, Inc.. p. 156. ISBN 0-13-600663-9.

[17] Jaehwan Lee, Pete Keleher and Alan Sussman. Decentralized Dynamic Scheduling across Heterogeneous Multi-core Desktop Grids. In Proceedings of the 19th International Heterogeneity in Computing Workshop (HCW2010). April 2010. IEEE Computer Society Press. Appears with Workshop Proceedings of IPDPS 2010.