

Wave Scheduler: Scheduling for Faster Turnaround Time in Peer-based Desktop Grid Systems

Dayi Zhou and Virginia Lo
Computer and Information Science Department
University of Oregon
dayizhou|lo@cs.uoregon.edu

Abstract

The recent success of Internet-based computing projects, coupled with rapid developments in peer-to-peer systems, has stimulated interest in the notion of harvesting idle cycles under a peer-to-peer model. The problem we address in this paper is the development of scheduling strategies to achieve faster turnaround time in an open peer-based desktop grid system. The challenges for this problem are two-fold: How does the scheduler quickly discover idle cycles in the absence of global information about host availability? And how can faster turnaround time be achieved within the opportunistic scheduling environment offered by volunteer hosts? We propose a novel peer-based scheduling method, Wave Scheduler, which allow peers to self organize into a timezone-aware overlay network using a structured overlay network. The Wave Scheduler then exploits large blocks of idle night-time cycles by migrating jobs to hosts located in night-time zones around the globe, which are discovered by scalable resource discovery methods.

Simulation results show that the slowdown factors of all migration schemes are consistently lower than the slowdown factors of the non-migration schemes. Compared to traditional migration strategies we tested, the Wave Scheduler performs best. However under heavy load conditions, there is contention for those night-time hosts. Therefore, we propose an adaptive migration strategy for Wave Scheduler to further improve performance.

I. Introduction

It is widely acknowledged that a vast amount of idle cycles lie scattered throughout the Internet. The recent

success of Internet-based computing projects such as SETI@home [1] and the Stanford Folding Project [2], coupled with rapid developments in peer-to-peer systems, has stimulated interest in the notion of harvesting idle cycles for desktop machines under a peer-to-peer model.

A peer-based desktop grid system allows cycle donors to organize themselves into an overlay network. Each peer is a potential donor of idle cycles as well as a potential source of jobs for automatic scheduling in the virtual resource pool. Many current research projects are exploring Internet-wide cycle-sharing using a peer-based model [3], [4], [5], [6].

The major benefits of peer-based desktop grid systems are that they are scalable and lightweight, compared with institutional-based Grid systems [7], [8], [9], load sharing systems in local networks [10], [11], [12], [13] and Internet-based global computing projects [1], [2], [14], [15], [16]. The latter systems do not scale well because they depend on centralized servers for scheduling and resource management and often incur overhead for negotiation and administration.¹

However, to design scheduling methods satisfying jobs with fast turnaround requirements is a big challenge in dynamic, opportunistic peer-based desktop grid systems, which faces a number of unique challenges inherent to the nature of the peer-to-peer environment.

The challenge comes from the opportunistic and volatile nature of the peer-based desktop grid systems. Peer-based desktop grid systems use non-dedicated machines in which local jobs have much higher priority than foreign jobs. Therefore, compared to running on dedicated machines, the foreign job will make slower progress since it can only access a fraction of the host's CPU availability. The resources are highly volatile in peer-based desktop grid

¹A range of research issues faced by desktop grid systems are beyond the scope of this paper including incentives and fairness, security (malicious hosts, protecting the local host), etc.

system. Nodes may leave and join the system at any time, and resource owners may withdraw their resources at any time. Therefore, the foreign jobs may experience frequent failures due to the volatility of the resources.

Another challenge for design of an efficient scheduling system for peer-based desktop grid systems comes from the difficulties in collecting global and accurate resource information. It is unscalable to collect resource information on all the nodes in a large scale peer-based desktop grid system. Also users, especially home machine cycle donors, may find it is intrusive to report their CPU usage periodically to some remote clients. Therefore, scheduling in large scale cycle sharing systems are usually best effort scheduling based on limited resource information.

The problem we address in this paper is the development of scheduling strategies that achieve fast turnaround time and are deployable within a peer-based model. To our best knowledge, we are the first to study this scheduling problem in a fully distributed peer-based environment using Internet-wide cycle donors.

We propose *Wave Scheduler*, a novel scalable scheduler for peer-based desktop grid system. Wave scheduler has two major components: a self-organized, timezone-aware overlay network and an efficient scheduling and migration strategy.

Self-organized timezone-aware overlay network. The Wave Scheduler allows hosts to organize themselves by timezone to indicate when they have large blocks of idle time. The Wave Scheduler uses a timezone-aware overlay network built on a structured overlay network such as CAN [17], Chord [18] or Pastry [19]. For example, a host in Pacific Time timezone can join the corresponding area in the overlay network to indicate that with high probability his machine will be idle from 8:00-14:00 GMT when he sleeps.

Efficient scheduling and migration. Under the Wave Scheduler, a client initially schedules its job on a host in the current nighttime zone. When the host machine is no longer idle, the job is migrated to a new nighttime zone. Thus, jobs ride a wave of idle cycles around the world to reduce turnaround time.

A class of applications suitable for scheduling and migration in Wave Scheduler are long running *workpile* jobs. Workpile jobs, also known as *bag-of-tasks*, are CPU intensive and embarrassingly parallel. For workpile jobs which run in the order of hours to days, the overheads of migration costs are negligible. Examples of such workpile applications include state-space search algorithms, ray-tracing programs, and long-running simulations.

The contributions of this paper include the following:

- Creation of an innovative scheduling method, the Wave scheduler, which exploits large chunk of idle cycles such as the nighttime cycles for fast turnaround.
- Analysis of a range of migration strategies, including migration under Wave Scheduler, with respect to turnaround time, success rate, and overhead.

II. Problem Description

The problem we address in this paper is the design of scheduling strategies for faster turnaround time for bag-of-tasks applications in large, open peer-based desktop grid systems. In this section, we discuss the key dimensions of the problem including the open peer-based cycle sharing infrastructure, the host availability model, and the characteristics of the applications supported by our scheduling strategies.

A. Open peer-based desktop grid system

The peer-based desktop grid system we are studying is open, symmetric, and fully distributed. In peer-based desktop grid systems, hosts join a community-based overlay network depending on their interests. Any client peer can submit applications; the application scheduler on the client will select a group of hosts whose resources match requirements of the application.

In a large, open peer-based desktop grid system, global resource discovery and scheduling of idle hosts using centralized servers is not feasible. Several research projects have addressed resource discovery for peer-to-peer cycle sharing [20], [4], [21]. One approach builds an auxiliary hierarchy among the peers such that a dynamic group of super-peers (similar to Gnutella's ultrapeers [22]) are designated to collect resource information and conduct the resource discovery on behalf of the other peers. Alternatively, each client peer uses a distributed scalable algorithm to discover available resources. These protocols are either probing based, such as expanding ring and random walk, or gossip based such as exchanging and caching resource advertisements.

On receiving a request for computational cycles from some client, the host returns resource information including CPU power, memory size, disk size, etc. It also returns information regarding the current status of the machine regarding its availability to accept a foreign job. The client then chooses a host to schedule the job using its host selection criteria and waits for acknowledgment. The client then ships the job to the selected host. The job can be migrated to a new host if the current host becomes unavailable.

B. Host availability model

In open cycle sharing systems, users can make strict policies to decide when the host is available, which will limit the amount of cycles available to foreign jobs. A lesson learned from previous cycle sharing systems is that inconvenienced users will quickly withdraw from the system. To keep the users in the system, a cycle sharing system must be designed to preserve user control over her idle cycles and to cause minimal disturbance to the host machine.

The legendary Condor load sharing project [10], [8], developed at the University of Wisconsin in the 1980s for distributed systems, allows users to specify that foreign jobs should be preempted whenever mouse or keyboard events occur. Condor also supports strict owner policies in its *classified advertisement* of resource information [23]: users can rank foreign applications, specify a minimum CPU load threshold for cycle sharing, or specify specific time slots when foreign jobs are allowed to that host.

Another example of strict user-defined policies is the popular SETI@home project [1]. Users donate cycles by downloading a screensaver program from a central SETI@home server. The screensaver works like standard screensaver programs: it runs when no mouse or keyboard activities have been detected for a pre-configured time; otherwise it sleeps. SETI@home can also be configured to run in background mode, enabling it to continue computing all the time. However, screensaver mode is the default mode suggested by the SETI@home group and is the dominant mode employed by the SETI@home volunteer community.

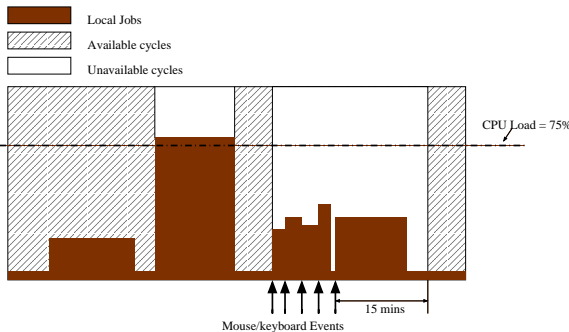


Fig. 1. A sample host profile of available idle cycles

As peer-based cycle sharing becomes more widespread, the majority of the users will most likely be more conservative than current users when donating cycles to anonymous clients in an open cycle sharing environment. The cycle sharing pattern will be cautiously generous: users are willing to join the system only if they know their own work will not be disturbed by foreign jobs. CPU cycle

sharing will be likely limited to the time when owners are away from their machines and the CPU load from local applications is light. Figure 1 illustrates a sample host profile of available idle cycles under a strict user local policy: Host is available only when CPU load is less than 75% and there is no mouse or keyboard activity for 15 minutes.

C. Application characteristic

The type of applications we are looking at in this study are large *Workpile (Bag-of-tasks)* jobs, requiring large amounts of CPU cycles but little if any data communication. Examples of workpile applications include state-space search algorithms, ray-tracing programs, gene sequencing and long-running simulations. Often these applications have higher performance requirements such as faster turnaround time and higher throughput. Some of the above applications may have real time constraints, such as weather forecasting, or deadlines such as scientific simulations that must be completed in time for a conference submission (such as JSSPP).

The migration cost is higher in global peer-based cycle sharing systems than in local area network as the codes and data are transferred on the Internet. If a short job that runs for a few minutes, is migrated many times in its life span, the accumulated migration cost may well counter the migration benefit. Long jobs which run for hours or even for months receive maximal benefit from migration schemes. For such jobs, the cost of migration, which includes resource discovery overhead to find a migration target, checkpointing, and cost to transfer the code and data is negligible compared to the total runtime of the job.

The size of many long running applications is small and these applications require minimal data communication. For example, the average data moved per CPU hour by users of SETI@home is only 21.25 KB, which is feasible even for users with slow dial-up connections. With respect to program size, Stanford Folding is only about 371KB, and SETI@home is around 791KB (because it includes the graphical interface for the screensaver). These applications run for a long time. Using the same SETI@home example, the average computation time of each job is over 6 hours (the major type of CPU in SETI@home is Intel x86).

III. Wave scheduling for workpile applications

Wave scheduling is an idea that springs naturally from the observation that millions of machines are idle for large chunks of time. For example, most home machines and office machines lie idle at night. It is influenced by the notion of prime time v. non-prime time scheduling regimes

enforced by parallel job schedulers [24], which schedules long jobs at night to improve turnaround time.

There are many motivations for the design of Wave Scheduler.

- First, resource information such as when the host will be idle and how long the host will continue to be idle with high probability will help the scheduler make much better decisions. Wave scheduler builds this information into the overlay network by having hosts organize themselves into the overlay network according to their timezone information.
- Second, efficient use of large and relatively stable chunks of continuing idle cycles provides the best performance improvement, while performance improvement by using sporadic and volatile small pieces of idle cycles in seconds or minutes is marginal and may be countered by high resource discovery and scheduling overhead. Therefore, the Wave scheduler proposes to use long idle night-time cycles.
- Third, the cycle donors are geographically distributed nodes, so that their idle times are well dispersed on the human time scale. Machines enter night-time in the order of the time zones around the world. In such a system, migration is an efficient scheme for faster turnaround.
- Fourth, the scheduler should be easy to install and not intrusive to users' privacy. The particular wave scheduler studied in this paper only needs minimal user input such as time zone information.

Wave Scheduler builds a timezone-aware, structured overlay network and it migrates jobs from busy hosts to idle hosts. Wave scheduler can utilize any structured overlay network such as CAN [17], Pastry [19], and Chord [18]. The algorithm we present here uses a CAN overlay [17] to organize nodes located in different timezones. In this section, we introduce the structured overlay network, and then we describe Wave Scheduler.

A. Structured overlay network

Structured overlay networks take advantage of the power of regular topologies: symmetric and balanced topologies, explicit label-based or Cartesian distance based routing and theoretical-bounded virtual routing latency and neighbor table size. In this section, we will describe the original CAN (Content Addressable Network) protocol, which is used by our Wave Scheduler.

The CAN structured overlay [17] uses a Cartesian coordinate space. The entire space is partitioned among all the physical nodes in the system at any time, and each physical node owns a distinct subspace in the overall space. Any coordinate that lies within its subspace can be used as an address to the physical node which owns the

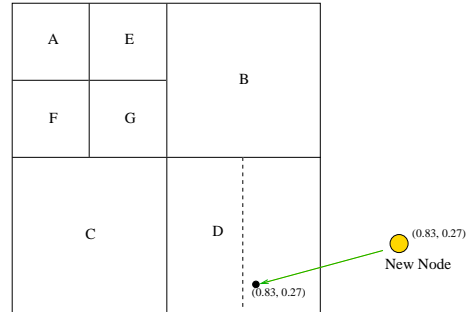


Fig. 2. Node join in CAN

subspace. For example, in Figure 2, the whole Cartesian space is partitioned among 7 nodes, A, B, C, D, E, F and G. The neighbors of a given node in the CAN overlay are those nodes who are adjacent along $d - 1$ dimensions and abut along one dimension. The neighbors of node G are nodes F, E, B, and C.

In Figure 2, new node N joins the CAN space by picking a coordinate in the Cartesian space and sending a message into the CAN destined for that coordinate. (The method for picking the coordinate is application-specific.) There is a bootstrap mechanism for injecting the message into the CAN space at a starting node. The message is then routed from node to node through the CAN space until it reaches node D who owns the subspace containing N's coordinate. Each node along the way forwards the message to the neighbor that is closest in the Cartesian space to the destination. When the message reaches D, it then splits the CAN space with the new node N and adjusts the neighbor tables accordingly. Latency for the join operation is bounded by $O(n^{1/d})$ in which n is the number of peers in the overlay network and d is the number of dimensions of the CAN overlay network.

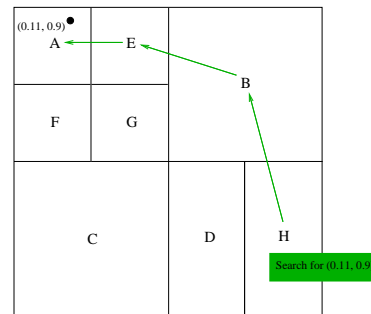


Fig. 3. Routing in CAN

Figure 3 illustrates the coordinate-based routing from a source node to a destination node which uses the same hop by hop forwarding mechanism and is also bounded by $O(n^{1/d})$.

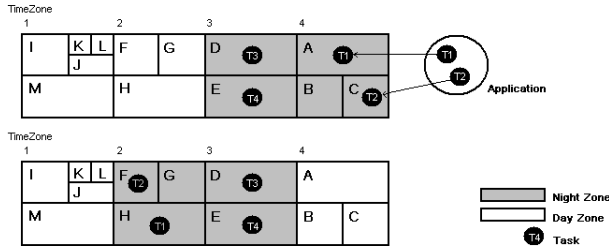


Fig. 4. Job initiation and migration in wave scheduling

In this study, we utilize CAN’s label-based routing for host discovery in all of our scheduling strategies. The CAN protocol is fully described by Ratnasamy et.al [17].

B. Wave Scheduling

In this section, we will present the Wave Scheduler, which takes advantage of the idle night cycles. Our wave scheduling protocol functions as follows (see Figure 4).

- **Wavezones in the CAN overlay.** We divide the CAN virtual overlay space into several *wavezones*. Each *wavezone* represents several geographical timezones. A straightforward way to divide the CAN space is to select one dimension of the d-dimensional Cartesian space used by CAN and divide the space into several wavezones along that dimension. For example, a 1 x 24 CAN space could be divided into 4 wavezones each containing 6 continuous timezones.
- **Host nodes join the overlay:** A host node that wishes to offer its night-time cycles knows which timezone it occupies, say timezone 8. It randomly selects a node label in wavezone 2 containing timezone 8 such as (0.37, 7.12) and sends a join message to that node. According to the CAN protocol, the message will reach the physical node in charge of CAN node (0.37, 7.12) who will split the portion of the CAN space it owns, giving part of it to the new host node.
- **Client selects initial nightzone:** The scheduler for a workpile application knows which timezones are currently nightzones. It select one of these nightzones (based on some nightzone selection criteria) and decides on the number h of hosts it would like to target.
- **Selects set of target hosts:** The scheduler randomly generates a collection of h node labels in the wavezone containing the target nightzone and sends a request message to each target node label using CAN routing which finds the physical host which owns that node label. Or it does an expanding ring search starting from a random point in the target wavezone. After negotiations, the application scheduler selects a

subset of those nodes to ship jobs to.

- **Migration to next timezone:** When morning comes to a host node and the host is no longer available, it selects a new target nightzone, randomly selects a host node in that nightzone for migration, and after negotiating with that host, migrates the unfinished job to the new host.

C. Extensions to Wave Scheduler

The *night-time* concept can be extended to any long interval of available time. The overlay does not necessary need to be timezone based but can be organized based on available time intervals. For example, a user in Pacific Time timezone can register her home machine in a wavezone containing hosts idle from 0:00-10:00 GMT, when she knows she is at work during the daytime. Wave scheduler can even accept more complicated user profiles, which indicates users’ daily schedules.

Wave Scheduling can also be easily leveraged to handle heterogeneity of the hosts in the system. Information such as operating systems, memory and machine types can be represented by further dividing the node label space or adding separate dimensions. For example, a third dimension is added to represent the operating system and it is divided into subspaces that represent each type of operating systems. When a client needs extra cycles, it can generate a node label. The first two dimensions are generated in the way described above and the third dimension has a value indicating the hosts need to be running a specific operating system. The third dimension can be empty, indicating the client does not care about the operating system type. In heterogeneous peer-based desktop grid system, the hosts have different CPU speeds. The difference between CPU speeds can be solved by normalizing the length of the idle time by the CPU speed and organize hosts according to this normalized profile information.

IV. Peer-based scheduling strategies that utilize migration

In this section, we describe the scheduling/migration strategies we evaluated in this paper. First we describe the key components that serve as building blocks for our scheduling strategies. Then we describe the peer-based migration strategies studied in this paper. In this study, we assume that all the migration schemes are built on a structured overlay network.

Our scheduling model is composed of the following four key components: host selection criteria, host discovery strategy, local scheduling policy, and migration scheme. When a client wants to schedule a job, the scheduler chooses the candidate host(s) satisfying the host selection

criteria via host discovery. Then it schedules the job on the candidate host. If there are multiple candidate hosts, it will select one to schedule the job on. A migration scheme decides when and where to migrate the job. It uses host discovery and host selection strategy to decide where to migrate the host. Other modules including checkpointing, result verification and monitoring are out of the scope of this discussion.

Host selection A client uses its host selection criteria to decide whether the host can be a candidate, and it selects one of them on which to schedule a job if there are multiple candidates. We also assume that each host can host at most one foreign job at a time.

The following terms define the criteria we use in this study, motivated by our earlier discussion of strict user control. *Unclaimed* means that there is no foreign job on that host. *Available* means that there is no foreign job on that host and the host is idle. Local user policy can be made to decide whether the host is idle based on criteria such as if the CPU load is below some threshold, or if there are no recent mouse/keyboard activities. The user policy can even blackout arbitrary time slots.

Different scheduling methods use different host selection criteria. Simple scheduling methods relax their hosts selection criteria to use any *unclaimed* hosts, while fast turnaround scheduling methods try to schedule foreign job on *available* hosts for instant execution.

In this study, we use a low-complexity host selection strategy when there are multiple candidates: a client selects the first discovered host that satisfies the particular host selection criteria used by the scheduling method.

Host Discovery. The purpose of the host discovery scheme is to discovery candidate hosts to accept the foreign job. The scheme needs to take into account the tradeoff between the message overhead and the success rate of the operation [20], [21]. Two schemes are used in this study.

Label-based random discovery. When the client needs extra cycles, the client randomly chooses a point in the CAN coordinate space and sends a request to that point. The peer owning that point will receive this request and return the resource information about whether it has already accepted a foreign job (*claimed*) and whether it is *available*. If the host does not satisfy the host selection criteria, the client can repeatedly generate another random point and contact another host. The maximum number of queries the client can issue is a configurable parameter.

Expanding ring search. When the client needs extra cycles, the client sends out a request with the host selection criteria to its direct neighbors. On receiving

such request, if the criteria can be satisfied, the neighbor acknowledges the client. If the request is not satisfied, the client increases the search scope and forwards the request to its neighbors one-hop farther away. This procedure is repeated until the request is satisfied or the searching scope limit is reached. The client can choose to initiate the search in its own neighborhood, or it can choose a random point in the system (by generating a random node label and asking the owner of that random label to start the search). The benefit of the latter approach is to create a balanced load in cases of a skewed client request distribution in the CAN space.

Local Scheduling. The local scheduling policy on a host determines the type of service a host gives to a foreign job that it has accepted. Two common policies are: *screensaver* and *background*. With screensaver, foreign jobs can only run when there is no recent mouse/keyboard activity. With background, foreign jobs continue running as background processes with low priority even when users are present at their machines. We only consider the screensaver option in this study to reflect a conservative policy most likely in open peer-to-peer cycle sharing systems.

Note that under screensaver mode, the availability of a host to run the foreign job does not mean that the job receives 100% of the machine's CPU cycles. Instead the job concurrently shares cycles with other local jobs.

Migration: Migration was originally designed for load sharing in distributed computing to move active processes from a heavily loaded machine to a lightly loaded machine. Theoretical and experimental studies have shown that migration can be used to improve turnaround time [25], [26].

There are two important issues for migration schemes: *when* to migrate the jobs and *where* to migrate the jobs. Traditional load sharing systems used central servers or high overhead information exchange to collect resource information about hosts in the system to determine when and where to migrate jobs [25], [26]. New scalable strategies are needed to guide migration decisions in a peer-to-peer system.

The optimal solution of *when* to migration the job requires accurate predication of future resource availability on all the hosts. Many researchers have addressed the CPU availability prediction problem for the Grid or for load sharing systems [27], [28], [29], but they all require a central location to collect and process the resource information. In our study, we assume there is no resource availability prediction and that migration is a simple best effort decision based primarily on local information, e.g. when the host becomes unavailable due to user activity.

The same resource availability issue exists for *where*

to migrate the job. But the issue of *where* to migrate the job is also related to the scalable host discovery which we have discussed. The scheduler needs the host discovery to discover candidate hosts which are suitable to migrate the job to.

We compare several migration schemes that differ regarding when to migrate and where to migrate.

The options for when to migrate include:

- *Immediate migration.* Once the host is no longer available, the foreign jobs are immediately migrated to another available host.
- *Linger migration.* Linger migration allows foreign jobs to linger on the host for a random amount of time after the host becomes unavailable. After lingering, if the host becomes available again, the foreign job can continue execution on that host. Linger migration avoids unnecessary migration as the host might only be temporarily unavailable. Linger migration can also be used to avoid network congestion or contention for available hosts when a large number of jobs need to be migrated at the same time.

There are also two options for where to migrate the jobs:

- *Random.* The new host is selected in a random area in the overlay network. There is no targeted area for the search; the new host is a random host found in the area where the resource discovery scheme is launched.
- *Night-time machines.* The night-time machines are assumed to be idle for a large chunk of time. The Wave Scheduler uses the geographic CAN overlay to select a host in the night-time zone.

A. Scheduling strategies

The scheduling strategies we study are built on the above components. They all use the same host discovery schemes but different host selection criteria and different migration schemes.

Each strategy has two distinct steps: initial scheduling and later migration. In initial scheduling, the initiator of the job uses host discovery to discover hosts satisfying the host selection criteria and schedules job on the chosen host. The migration schemes also use host discovery to discover candidate hosts, and they use different options discussed above to decide when and where to migrate the job. Table I summarizes the difference between different migration schemes.

The non-migration strategy follows the SETI@home model. It uses the more relaxed host selection criteria: any *unclaimed* host can be a candidate.

- **No-migration:** With no-migration, a client initially schedules the task on an unclaimed host, and the

	When to migrate	
Where to migrate	Immediate Migration	Linger
Random Host	Migration-immediate	Migration-linger
Host in night-zone	Wave-immediate	Wave-linger

TABLE I. Different migration strategies

task never migrates during its lifetime. The task runs in screensaver mode when the user is not using the machine, and sleeps when the machine is unavailable.

The following are all migration schemes. The first four migration schemes try to only use *available* host for fast execution. When it fails to find *available hosts* for migration, the host will inform the initiator of the job and let the initiator reschedule the job.

- **Migration-immediate:** With migration-immediate, the client initially schedules the task on an available host. When the host becomes unavailable migration-immediate immediately migrates the job to a *random* available host. In the best case, the task begins running immediately, migrates as soon as the current host is unavailable, and continues to run right away on a new available host.
- **Wave-immediate:** Wave-immediate works the same as migration-immediate except the job is migrated to a host in the night-time zone.
- **Migration-linger:** With migration-linger, a client initially schedules the task on an available host. When the host becomes unavailable, migration-linger allows the task to linger on the host for a random amount of time. If the host is still unavailable after the lingering time is up, it then migrates.
- **Wave-linger:** Wave-linger works the same as migration-linger except that the job is allowed to linger before migrating to a host in the night-time zone.

The migration schemes described above put minimal burden on the hosts. A host only needs to try to migrate the task once when it become unavailable, i.e. there is no backoff and retry. Instead, if the host fails to find an idle host, it notifies the original client node who initially scheduled the job on this host, letting the client reschedule the job.

The last two migration strategies are more persistent in their efforts to find an available host. They are adaptive strategies in that they adjust to the conditions on the local host, and on their ability to find a migration target. These adaptive strategies put a bigger burden on the host since it must retry several times on behalf of the foreign task.

- **Migration-adaptive:** For initial scheduling, migration-adaptive tries to find a host that is available. If it cannot, migration-adaptive schedules

the task on an unclaimed host where the task will sleep for a random amount of time.

When the host becomes unavailable, migration-adaptive will try to migrate the task to a *random* new host that is available. If it cannot find such a host, it allows the job to linger on the current host for a random amount of time and try again later. A cycle of attempted migration and lingering is repeated until the job finishes.

- **Wave-adaptive:** Wave-adaptive is the same as migration-adaptive except that it migrates to a host in the night-time wave zone.

V. Simulation

We conducted simulations to investigate the performance of the migration strategies described above and their effectiveness at reducing turnaround time, relative to a no-migration policy similar to SETI@home. We also evaluated the performance of the Wave Scheduler to see what gains are achievable through our strategy of exploiting knowledge of available idle blocks of time at night.

A. Simulation configuration

We use a 5000 node structured overlay in the simulation. Nodes join the overlay following the CAN protocol (or timezone-aware CAN protocol in the case of the Wave scheduler). The simulation is built with ns, a widely used network simulation tool [30].

Profile of available cycles on hosts. To evaluate the performance of different scheduling methods, a coarse-grain hourly synthetic profile is generated for each machine as follows: During the night-time (from 12pm to 6 am), the host is available with a very low CPU utilization level, from 0% to 10%. During the daytime, for each one hour slot it is randomly decided whether the machine is available or not. Finally, the local CPU load in a free daytime slot is generated from a uniform distribution ranging from 0% to 30%. We assume that when a host is running a foreign job, it can still initiate resource discovery for migration and relay messages for other hosts. The percentage of available time during the day varies from 5% to 95%. For simplicity, we assume all the hosts have the same computational power.

Job workload. During the simulation, a given peer can be both a client and a host. A random group of peers (10% to 90%) are chosen as clients. Each client submits a job to the system at a random point during the day. The job *runtime* is defined as the time needed for the job to run to completion on a dedicated machine. Job runtime is randomly distributed from 12 hours to 24 hours.

Host discovery parameters. We set the parameters of the resource discovery schemes to be scalable and to have low latency. The maximum number of scheduling attempts for random node label-based resource discovery is 5 times and the search scope for expanding ring search is 2 hops.

Migration parameters. The lingering time for the linger-migration models is randomly chosen in the range 1 hour to 3 hours. In the adaptive model, the linger time of a foreign job when it cannot find a better host to migrate to is also randomly chosen in the range 1 hour to 3 hours.

Wave scheduler. For the Wave scheduler, a 1x 24 CAN space is divided into 6 wavezones, each containing 4 time zones based on its second dimension. We studied the performance of a variety of strategies for selecting the initial wavezone and the wavezone to migrate to. The variations included (a) migrate the job to the wavezone whose earliest timezone just entered night-time, (b) migrate the job to a random night-time zone, and (c) migrate the job to a wavezone that currently contains the most night-time zones. The first option performs better than the others, since it provides the maximal length of night-time cycles. The simulation results presented in this paper use this option. However, it may be better to randomly select a nightzone to balance network traffic if many jobs simultaneously require wave scheduling.

B. Simulation Metrics

Our evaluation of different scheduling strategies is focused on the turnaround time of a job, the time from when it first began execution to when it completes execution in the system.

In our study, a job is considered to have failed if the client fails to find a host satisfying host selection criteria, either when it initially tries to schedule the job, or when it later tries to migrate. Most of the performance metrics are measured only for those jobs that successfully complete execution. In this study, we do not model rescheduling, as we are interested in the success rate of the first scheduling attempt which includes the initial scheduling and the following successful migrations. The job completes in the shortest time if the client only needs to schedule the job once, so the slowdown factor measured this way show the peak performance of each migration scheme. Also it is interesting to see what percentage of jobs needs to be rescheduled under different migration models.

The metrics used in the study are the followings:

- **% of jobs that fail to complete (job failure rate):** the number of failed jobs divided by the total number of jobs submitted to the system.
- **Average slowdown factor:** The slowdown of a job is its turnaround time (time to complete execution in the peer-to-peer cycle sharing system) divided by the job

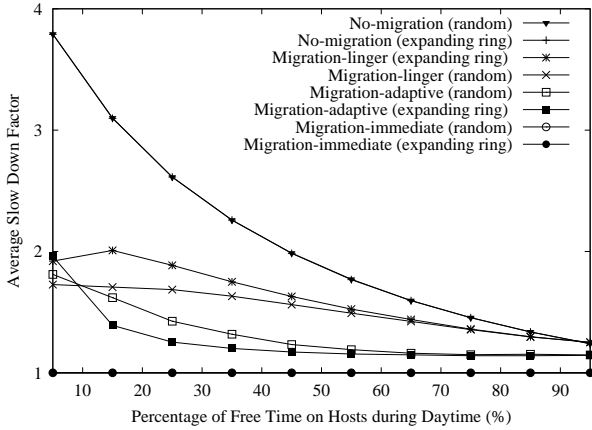


Fig. 5. Average slowdown factor for no-migration vs. migration (The percentage of clients in the system is 20%)

runtime (time to complete execution on a dedicated machine). We average the slowdown over all jobs that successfully complete execution.

- **Average number of migrations per job:** the number of times a job migrates during its lifetime in the system, averaged over all jobs that successfully complete execution.

We do not include migration and resource discovery overhead when plotting the average slowdown factor. The migration and resource discovery overhead do not make a visible difference in the results when migrations and resource discoveries are infrequent and the jobs run for a long time. We will analyze migration overhead, which dominates the computation overhead in the discussion of number of migrations.

C. Simulation Results

In this section, the legends in each graphs are ordered from top to bottom to match the relative position of the corresponding curves. Each data point is the average over 15 simulation runs.

1) *No-migration vs. migration:* We first compare no-migration with the two basic migration schemes: migration-immediate and migration-linger. We measure the performance of these scheduling strategies as a function of percentage of free time on the hosts. When the percentage of free time on hosts increases on the x-axis, the load of the system decreases. We also examine the impact of the resource discovery scheme employed.

(a) *The impact of migration on job turnaround times*

Figure 5 shows the average slowdown factor for successfully completed jobs as a function of free time on the hosts during the daytime hours. As expected, jobs

progress faster with more available time on hosts during daytime. The performance of the no-migration strategy is clearly the worst since there is no effort made to avoid long waiting times when the host is not free. Note that the slowdown of migration-immediate (for both expanding ring and random host discovery) is always 1, since only jobs that successfully run to completion are considered. The success rate of different scheduling schemes will be discussed in section V-C.1(b).

The performance of migration-linger is better than the no-migration strategy but worse than the others with its wasted lingering time associated with each migration. The performance of the adaptive models is the closest to the idealized migration-immediate schemes since it adaptively invokes the migration-immediate scheme whenever possible.

The slowdown factor is mainly influenced by the migration model used. However, for the linger and adaptive strategies, the resource discovery protocol also plays a role when the free time on the host is limited (e.g. when the percentage of hosts free time during daytime is less than 65%). We noticed that for the linger strategy, random performs better with respect to slowdown, but for the adaptive strategy, expanding ring performs better. This can be explained as follows: For comparable search times, expanding ring contacts more hosts than the random node label-based search, and therefore yields a higher successful migration rate. However, since with migration-linger, every successful migration implies (wasted) lingering time, ultimately random has lower slowdown with its lower migration success rate. This observation is supported by Figure 9 which shows the average number of migrations for each strategy. For the adaptive strategy, expanding ring has lower slowdown than random as expected.

Figure 5 also shows that the slowdown factor for the no-migration strategy and for migration-immediate is insensitive to the resource discovery schemes.

Figure 6 further confirms the improvement of turnaround time when using a migration model under heavy load. The majority of jobs scheduled by no-migration scheduling experienced a slowdown greater than 2 and in the extreme case, jobs may experience slowdown greater than 5. The majority of jobs scheduled by migration-adaptive and migration-immediate have small slowdown or no slowdown at all.

(b) *The impact of migration on successful job completion*

The above results regarding slowdown factor cannot be considered in isolation. In particular, it is necessary to also consider the job failure rates, i.e. percentage of jobs for which a host satisfying host selection criteria cannot be found either in initial scheduling or in migration.

Figure 7 shows the percentage of jobs that failed to

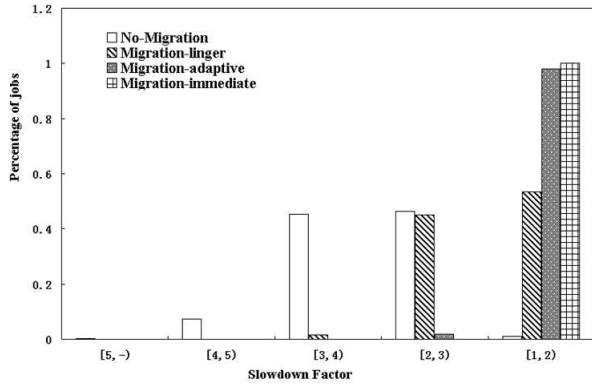


Fig. 6. histogram of slowdown factors of successfully finished jobs (The percentage of clients is 20% and the percentage of free time on hosts is 15%)

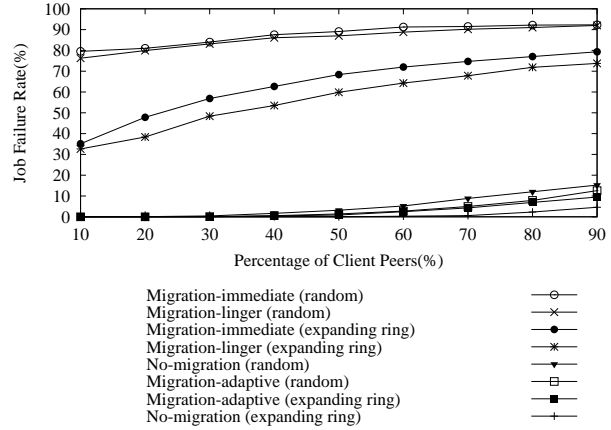


Fig. 8. % of jobs that fail to complete (The percentage of free time on the hosts during the day is 15%)

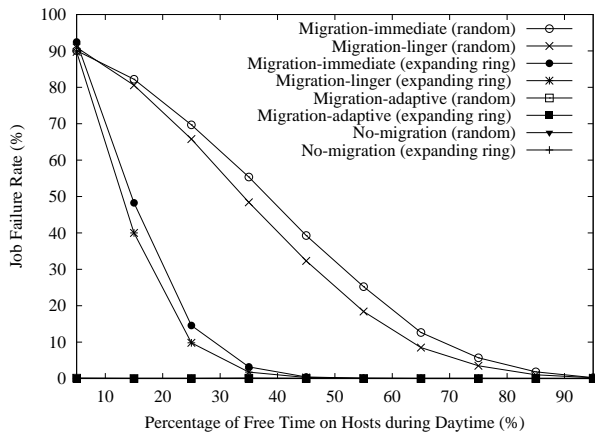


Fig. 7. % of jobs that fail to complete (The percentage of clients in the system is 20%)

finish assuming the same simulation configuration as in Figure 5. For the two strict migration models (migration-immediate and migration-linger) which only use local availability information, the failure rate is extremely high. The adaptive models, which use a small amount of global information at migration time, have dramatically fewer failed jobs – close to zero.

Clearly, there is a tradeoff between the job turnaround time and percentage of jobs that successfully complete. The strict models have the lowest turnaround time, but extremely high failure rates when free time on the hosts is limited. The adaptive model performs best because it achieves low turnaround time with most of the jobs successfully completed.

When the number of client requests increase, there will be intense competition for free hosts. When the free time on these hosts is small, the situation is even worse.

Figure 8 shows that the failure rate of all scheduling strategies increases with the increasing number of client requests. The persistently high failure rate of migration-immediate makes it impractical for real applications when the available resources are very limited.

The simulation results show that with abundant host free time, the failure rate of migration-adaptive using an expanding ring search is even slightly lower than the no-migration scheme.

(c) Number of migrations during job lifetime

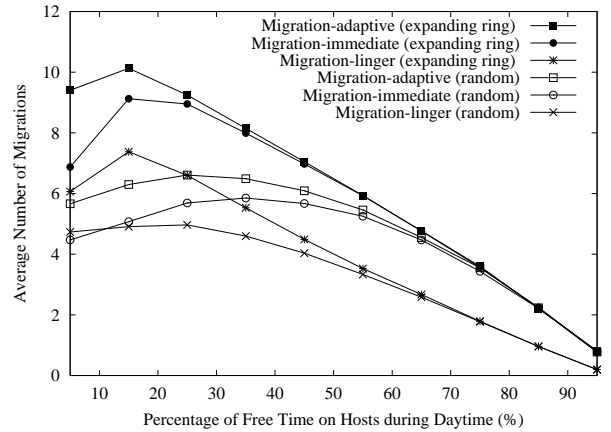


Fig. 9. Average number of migrations for successfully finished jobs (The percentage of clients in the system is 20%)

Figure 9 shows that the average number of migrations varies with the different migration scheduling strategies. The graph shows that when the percentage of host free time increases, the number of migrations increases first and then decreases. The reason is that there are two factors

that influence the number of migrations: the number of currently available hosts and the length of free time slots on the hosts. With more available hosts, there is higher chance of migration success and therefore a larger number of migrations. With longer free time slots, the need for the jobs to migrate is reduced. With higher percentage of free time, the amount of currently available hosts increases and the length of free time slots also increases.

We can demonstrate that migration overhead is low using the same graph. In early morning or late night, the network traffic in the Internet is usually light. Therefore the network connection from the end-host to the Internet is usually the bottleneck link when downloading or uploading data. In the following computation, we assume the upload bandwidth of the host is 256kb, which is the bandwidth of slow-end DSL users and download bandwidth is higher than upload bandwidth with DSL. If the amount of data to be transmitted during the migration is 1MB, the slowdown factor of migration schemes will increase by at most 0.005 when the running time of the job is longer than 12 hours. Even when the amount of data to transmit is 20MB, which is quite large for a scientific computation, the influence is at most 0.1. The time overhead of resource discoveries is much smaller and negligible compared with that of migration.

2) *Performance of Wave Scheduler:* This section presents the evaluation of the Wave Scheduler. In order to focus on the difference between migration strategies, we only describe results with the resource discovery method set as expanding ring. (Simulations with random node label-based discovery show the same relative performance.)

(a) *Impact of Wave scheduler on turnaround time*

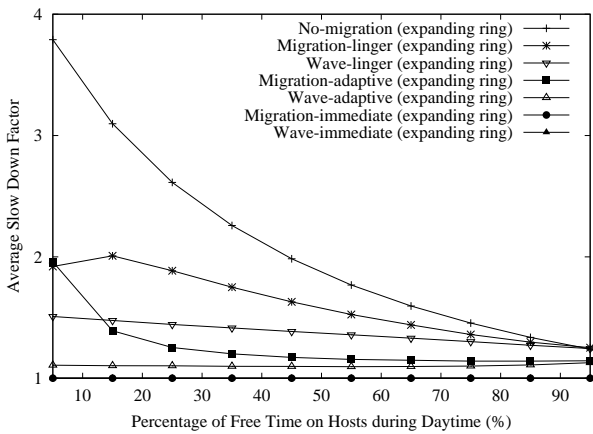


Fig. 10. Wave Scheduler: Average slow down factor(The percentage of client request is 20%)

Figure 10 shows that the turnaround time of jobs with Wave Schedulers is lower than other migration schedulers.

Jobs progress faster with the Wave Scheduler because it can identify hosts with potentially large chunks of available times. The turnaround time of wave-adaptive is consistently low, while the turnaround time of migration-adaptive is significantly higher when the amount of free time is small. When the percentage of free time on hosts is 15%, the turnaround time of jobs under wave-adaptive is about 75% of that under migration-adaptive.

(b) *Impact of Wave scheduler on successful job completion*

The percentage of jobs that fails to complete using the Wave scheduler is influenced by two factors. Wave identifies available hosts with large chunks of future free time. However, if the ratio of requests to the number of such hosts is limited, there will be scheduling conflicts.

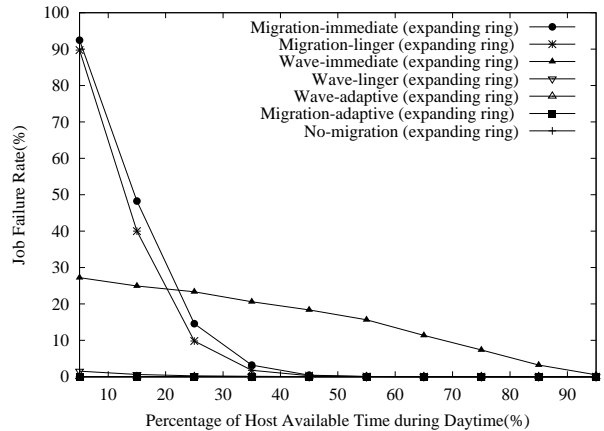


Fig. 11. Wave scheduler: % of job that fail to complete (The percentage of clients in the system is 20%)

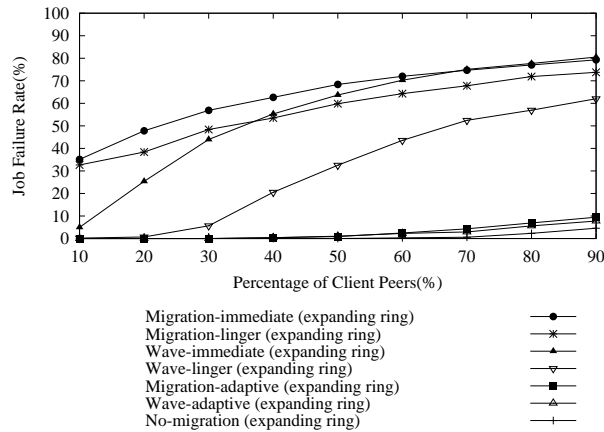


Fig. 12. Wave scheduler: % of job that fail to complete (The percentage of available time on the hosts during the day is 15%)

When the free time on hosts is limited, the Wave scheduler does better than other migration schemes since it was designed for this exact situation. (see Figure 12). The intensive contention for night-time hosts is relieved by wave-adaptive, which adapts to the amount of free time by continuing to stay on the hosts in case of contention. The job failure rate of wave-adaptive is competitive with the no-migration model and slightly lower than with migration-adaptive.

Figure 11 shows the percentage of jobs that failed to finish under the same simulation configuration as in Figure 10. The job failure rate of the Wave scheduler is relatively higher than others when the percentage of free time on hosts increases, as wave-immediate uses strict rules about using night-time hosts and this cause contention. The other two wave scheduler strategies perform as well as migration-adaptive and the no-migration strategy.

(c) Number of migrations during job lifetime with Wave scheduler

Figure 13 compares the average number of migrations of successfully finished jobs with the wave migration strategies versus the standard migration. As we expected, jobs scheduled with the Wave scheduler finished with fewer migrations, because it exploits the long available intervals at night while the others may end up using short, dispersed time slots during the day. As in the discussion about migration overhead in section V-C.1(c), the migration overhead of Wave Scheduler is even smaller compared with the standard migration schemes and therefore it is acceptable for jobs with long running time.

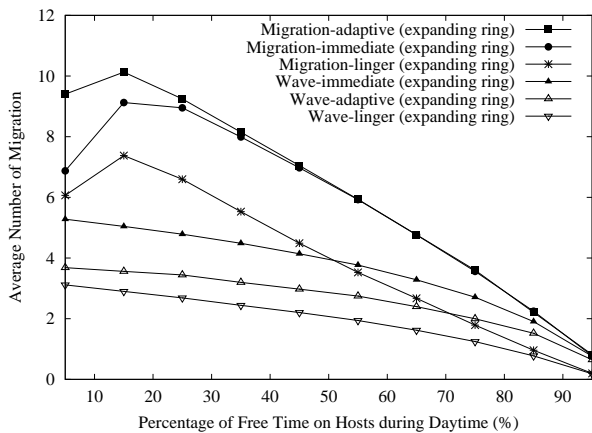


Fig. 13. Wave scheduler: Average number of migrations (The percentage of client in the system is 20%)

VI. Conclusion and Future Work

We studied job scheduling strategies with the goal of faster turnaround time in open peer-based cycle sharing systems. The fundamental lesson to be learned is that under strong owner policies on host availability for cycle sharing, the ability to utilize free cycles wherever and whenever they are available is critical. Migration achieves this goal by moving jobs away from busy hosts and to idle hosts. The best migration strategies that we studied, wave and adaptive, were able to use fairly straightforward mechanisms to make better decisions about when and where to migrate.

We also observed that careful design of the infrastructure of a peer-to-peer cycle sharing system impacts the performance of its schedulers. For example, the use of a structured overlay network that supports timezone-aware overlay was essential for the functioning of our wave scheduler. To recap:

- Compared with no-migration schemes, migration significantly reduces turnaround time.
- The adaptive strategies perform best overall with respect to both low turnaround time and low job failure rate.
- Wave scheduler performs better than the other migration strategies when the free time during the day is limited.
- Wave-adaptive improves upon wave because it reduces collisions on night-time hosts. It performs best among all the scheduling strategies.

To further improve the performance of Wave Scheduler and conduct a more realistic and comprehensive evaluation of Wave Scheduler and other peer-based scheduling using migration schemes, our ongoing and future work include the following tasks:

- 1) Improving wave scheduler by using the schemes discussed in section III including expanding the night-time cycles to any idle cycles and overlay construction including information other than just CPU cycles.
- 2) Evaluating Wave Scheduler and other migration schemes using a retry model in which clients retry after a random amount of time if the job fails to be scheduled on host in initial scheduling or migration. Alternatively, a client can make an intelligent decision about the how long it needs to wait before retry based on estimation of current load of the system.
- 3) Evaluating Wave Scheduler using workload trace such as Condor trace and desktop grid trace used by [31];
- 4) Studying the characteristics of bad-of-task scientific computation to understand what impact Wave Scheduler can make for real applications.
- 5) Collecting activity based resource profiles and generating CPU usage pattern from such profiles as a supporting

study to our work.

6) Studying the migration cost on an Internet test-bed such as PlanetLab [32].

A. Related Work

The related work can be divided into two categories: peer-to-peer networks and cycle sharing systems.

Peer-to-peer networks emerged with the popular file sharing systems. The first generation peer-to-peer protocols [22] were extended to efficient, scalable and robust structured peer-to-peer overlay networks [17], [19], [18]. Structured peer-to-peer overlay networks are motivated by distributed hash table algorithms which use consistent hash function to hash a key onto a physical node in the overlay network. A wealth of peer-to-peer applications including file sharing, storage sharing, and web caching are built atop structured overlay networks. The popularity of peer-to-peer file sharing techniques naturally stimulated the development of peer-based cycle sharing systems.

The second research area is cycle sharing systems which can be divided into three categories: Internet-based computing infrastructures, institutional-based cycle sharing systems and desktop Grid systems.

Internet-based computing infrastructures [1], [14], [15] use a client-server model to distribute tasks to volunteers scattered in the Internet. The hosts actively download tasks and data from a central server. A foreign task then stay on the same host during their entire life spans. The hosts report the results to the central server when the computation is done. Because Internet-based computing projects require manual coordination from central servers, they may experience downtime due to surges in hosts requests.

Institutional-based cycle sharing systems promote resource sharing within one or a few institutions. In order to access resources in institutional-based cycle sharing systems, the user first needs to acquire an account from the system administrator. The most notable practical institutional-based cycle sharing system is Condor [10], [8], which was first installed as a production system 15 years ago. The work continued to evolve from a load sharing system within one institution to a load sharing system within several institutions. Condor-G uses Globus [7] for inter-domain resource management. The strict membership requirement and heavyweight resource management and scheduling methods used by this type of system make it hard for average users to install and maintain their own cycle sharing systems.

The new desktop Grid systems [33], [34] harness idle cycles on desktop machines. Our work belongs to one type of desktop Grid system, the peer-based desktop Grid systems [3], [4], [5], which harness idle cycles on desktop

machines under a peer-based model. Each node in these systems can be either a single machine or an institution joining the peer-to-peer overlay network. Each peer can be both a cycle donor and a cycle consumer. Peer-based cycle sharing systems is a new research field, which charts many challenging research problems including scalable resource discovery and management, incentives for node to join the system, trust and security schemes. OurGrid [3] proposed an accounting scheme to aid equitable resource sharing in order to attract nodes to join the system. Flock of Condor [4] proposed to organize the nodes in a Pastry [19] overlay network. Nodes broadcast resource information in a limited scope for resource discovery. Our CCOF model [5] proposed a generic scalable modular peer-to-peer cycle sharing architecture which supports automatic scheduling for arbitrary client applications.

To our best knowledge, none of the previous work has addressed the fast turnaround scheduling problem in a scalable peer-based cycle sharing system. A recent paper [31] describes scheduling for rapid application turnaround on enterprise desktop grids. The computation infrastructure used a client-server model, in which a server stores the input data and schedules tasks to a host. When the host becomes available, it sends a request to the server. The server keeps a queue of available hosts and chooses the best hosts based on resource criteria such as clock rate and number of cycles delivered in the past. The work did not considering migration schemes. Moreover, this work is limited to scheduling within one institution and it uses a client-server infrastructure, while scheduling in a large scale fully distributed peer-based cycle sharing system is much more complicated and challenging.

References

- [1] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@home: An experiment in public-resource computing," *Communications of the ACM*, vol. 45, 2002.
- [2] "Folding@Home Distributed Computing <http://folding.stanford.edu/>."
- [3] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg, "OurGrid: An approach to easily assemble grids with equitable resource sharing," in *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, 2003.
- [4] A. Butt, R. Zhang, and Y. Hu, "A self-organizing flock of condors," in *Proceedings of SC2003*, 2003.
- [5] V. Lo, D. Zappala, D. Zhou, Y. Liu, and S. Zhao, "Cluster Computing on the Fly: P2P scheduling of idle cycles in the Internet," in *IPTPS*, 2004.
- [6] R. Gupta and A. Somani, "Compup2p: An architecture for sharing of computing resources in peer-to-peer networks with selfish nodes," in *Proceedings of second Workshop on the Economics of peer-to-peer systems*, 2004.
- [7] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, 1997.
- [8] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the grid," in *Grid Computing: Making the Global Infrastructure a Reality*,

F. Berman, G. Fox, and T. Hey, Eds. John Wiley & Sons Inc., 2002.

- [9] "Legion <http://www.cs.virginia.edu/legion/>."
- [10] M. Litzkow, M. Livny, and M. Mutka, "Condor -a hunter of idle workstations," in *the 8th International Conference on Distributed Computing Systems*, 1988.
- [11] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: a load sharing facility for large, heterogeneous distributed computer systems," *Software - Practice and Experience*, vol. 23, no. 12, 1993.
- [12] D. Ghormley, D. Petrou, S. Rodrigues, A. Vahdat, and T. Anderson, "GLUnix:a global layer unix of a network of workstations," *Software-Practice and Experience*, vol. 28, no. 9, 1998.
- [13] "LSF load sharing facility, <http://accl.grc.nasa.gov/lsf/aboutlsf.html>."
- [14] "BOINC: Berkeley open infrastructure for network computing, <http://boinc.berkeley.edu/>."
- [15] N. Camiel, S. London, N. Nisan, and O. Regev, "The popcorn project: Distributed computation over the Internet in java," in *Proceedings of The 6th International World Wide Web Conference*, 1997.
- [16] B. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff, "Charlotte: Metecomputing on the web," in *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.
- [17] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content addressable network," in *Proc. ACM SIGCOMM*, 2001.
- [18] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup service for internet applications," in *Proc. ACM SIGCOMM*, 2001.
- [19] A. Rowstron and P. Druschel, "Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proc. 18th IFIP/ACM Int'l Conf. on Distributed Systems Platforms*, 2001.
- [20] A. Iamnitchi and I. Foster, "A peer-to-peer approach to resource location in grid environments," in *Grid Resource Management*, J. Weglarz, J. Nabrzyski, J. Schopf, and M. Stroinski, Eds., 2003.
- [21] D. Zhou and V. Lo, "Cluster Computing on the Fly: resource discovery in a cycle sharing peer-to-peer system," in *Proceedings of the 4th International Workshop on Global and P2P Computing (GP2PC'04)*, 2004.
- [22] "The Gnutella protocol specification (version 0.6)."
- [23] R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed resource management for high throughput computing," in *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, 1998.
- [24] V. Lo and J. Mache, "Job scheduling for prime time vs. non-prime time," in *Proc 4th IEEE International Conference on Cluster Computing (CLUSTER 2002)*, 2002.
- [25] D. Eager, E. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. on Software Engineering*, vol. 12(5), 1986.
- [26] N. Shivaratri, P. Krueger, and M. Singhal, "Load distributing for locally distributed systems," *Computer*, vol. 25(12), 1992.
- [27] J. Brevik and D. Nurmi, "Quantifying machine availability in networked and desktop grid systems," UCSB, Tech. Rep. CS2003-37.
- [28] L. Yang, I. Foster, and J. Shopf, "Homeostatic and tendency-based CPU load predictions," 2003.
- [29] P. Dinda, "The statistical properties of host load," *Scientific Programming*, vol. 7, no. 3-4, 1999.
- [30] "ns, <http://www.isi.edu/nsnam/ns/>."
- [31] D. Kondo, A. Chien, and H. Casanova, "Resource management for rapid application turnaround on enterprise desktop grids," in *Proceedings of SC2004*, 2004.
- [32] "Planetlab, <http://www.planet-lab.org/>."
- [33] O. Lodygensky, G. Fedak, V. Neri, F. Cappello, D. Thain, and M. Livny, "Xtremweb & condor: sharing resources between internet connected condor pool," in *Proceedings of GP2PC2003(Global and Peer-to-Peer computing on large scale distributed systems)*, 2003.
- [34] "Entropia, inc. <http://www.entropia.com>."