GridBot: Execution of Bags of Tasks in Multiple Grids

Mark Silberstein, Artyom Sharov, Dan Geiger, Assaf Schuster

Technion—Israel Institute of Technology
{marks,sharov,dang,assaf}@cs.technion.ac.il

ABSTRACT

We present a holistic approach for efficient execution of bags-of-tasks (BOTs) on multiple grids, clusters, and volunteer computing grids virtualized as a single computing platform. The challenge is twofold: to assemble this compound environment and to employ it for execution of a mixture of throughput- and performance-oriented BOTs, *with a dozen to millions of tasks each. Our generic mechanism allows per BOT specification of dynamic arbitrary scheduling and replication policies as a function of the system state, BOT execution state, and BOT priority.*

We implement our mechanism in the GridBot *system and demonstrate its capabilities in a production setup. GridBot has executed hundreds of BOTs with over 9 million jobs during three months alone; these have been invoked on 25,000 hosts, 15,000 from the Superlink@Technion community grid and the rest from the Technion campus grid, local clusters, the Open Science Grid, EGEE, and the UW Madison pool.*

1. INTRODUCTION

Bags of Tasks (BOTs) are traditionally the most common type of parallel applications invoked in grids. BOTs are composed of a number of independent jobs that form a single logical computation. Their pleasantly parallel nature enables large scale invocation on the grids, despite slower networks, limited connectivity between geographically distributed resources, and job failures. Grid workflow engines have further strengthened the position of BOTs as the dominant type of grid workloads because they enable compound parallel applications with multiple interdependent BOTs [2,19,39].

Large grids, such as OSG [7] and EGEE [5], and *community grids* such as SETI@HOME have been very efficient in running *throughput-oriented* BOTs with thousands or millions of jobs. However, the invocation of moderate-sized, *performance-oriented* BOTs in large *non-dedicated* grids often results in higher turnaround times than executing them on a small *dedicated* cluster [25]. This is because shorter BOTs are more sensitive to the turnaround time of a single job. Their performance is dominated by the slowest job – even a single failure increases the turnaround time of the whole BOT. In contrast, in larger BOTs there are enough jobs to keep all the available CPUs busy for maximum available throughput to be achieved. Yet, the transition of a BOT from the *high-throughput phase* to the *tail phase*, characterized by the decrease in the number of incomplete jobs toward the end of the run, makes even throughput-oriented BOTs less immune to failures and delays.

The factors affecting the performance in large grids include the job invocation and scheduling overheads due to long queuing times and network delays. Job failures [24, 28] might also cause a job to be restarted from the beginning on another host. These failures often occur as a result of the opportunistic resource allocation mechanisms, which prematurely preempt lower priority jobs in favor of higher priority ones in order to maintain *fair-share* of grid resources among the users. Finally, grids typically trade job turnaround time for overall throughput. In community grids [13], for example, a batch of jobs is sent to a host for sequential execution, and the results are reported only after completion of all the jobs in the batch.

This throughput-optimized *modus operandi* of grid environments often makes them less attractive to scientists, who are tempted to build their own dedicated clusters optimized for shorter BOTs, instead of using the grids. However, the required computational demand typically outgrows the limited local resources, in particular if the scientific results prove successful. Thus, the same researchers will eventually need to access additional clusters, cloud computing infrastructures, institutional and international grids, and even end up establishing a community grid of their own.

Unfortunately, multiple separately managed grids without a common scheduling mechanism are an impediment to high performance for both shorter and larger BOTs. Static partitioning of a BOT among the grids does not account for sporadic changes in the resource availability, and reduces the number of jobs per BOT in each, decreasing overall efficiency. Thus, the segmentation of the resources requires dynamic job distribution and load-balancing.

Further complications arise if the workload comprises a mixture of large and small BOTs, as often happens in grid workflows. For example, better turnaround times will be obtained for smaller BOTs if they are scheduled on more reliable resources [25]. Routing BOTs to different grids according to the estimated BOT resource demand, as in the Grid Execution Hierarchy [31], results in rapid turnaround for smaller BOTs, but only for moderate system loads. Otherwise, the available resources become segmented and performance reduced for large and small BOTs alike. Also, any *static* policy that does not capture changes in the system state and BOT execution dynamics will be suboptimal. A BOT considered "throughput-oriented" at one point may become "performance-oriented" and vice versa, due to the changes in the computational demand of larger BOTs in the tail phase, and fluctuations in grid resource availability. Lastly, budget constraints, emerging in pay-

SC09 November 14-20, 2009, Portland, Oregon, USA.
Copyright 2009 ACM 978-1-60558-744-8/09/11 ...$10.00.

as-you-use cloud computing environments, may require a special resource allocation policy for some BOTs to reduce the costs.

Another aspect of multi-BOT scheduling is prioritization. For example, a shorter BOT will experience a significant slowdown in a FIFO queue if submitted after a long one. Consider also a scenario where the two BOTs are invoked by two different users contributing their own clusters to the system. Clearly the BOTs would be prioritized on the cluster belonging to the BOT owner, with lower priority on the foreign cluster. A simple priority queue, which would solve the problem of slowdown in the first scenario, will not suffice.

Contribution. We present a generic scalable mechanism for efficient concurrent execution of multiple arbitrary-sized BOTs in compound multi-grid environments. To the best of our knowledge, this is the first solution which combines several diverse grids in a single monolithic platform supporting flexible runtime policies for large-scale execution of multiple BOTs.

First, we unify the grids by establishing an overlay of *execution clients*, a technique termed *overlay computing* [3, 12, 29, 37]. While widely used for eliminating long queuing delays and aggregating multiple grids, the existing technologies fall short in grids with strict firewall policies and private networks. Our implementation overcomes this limitation while requiring no prior coordination with grid administrators, or deployment of additional software in the grids. Furthermore, community grid resources are integrated with all the others forming a unified work-dispatch framework.

Second, we apply several known techniques for achieving rapid turnaround of BOTs, including resource matching, job replication and dynamic bundling [34]. In particular, *replication* – speculative execution of multiple copies of the same job – was shown to decrease BOT turnaround time in failure-prone environments [14, 15, 25, 26, 38]. Many of these works devise specific replication algorithms applicable in a certain setup. Our contribution is in *the explicit separation of the* mechanisms *that implement these techniques from the* policies *that determine when and how the mechanisms are employed by the work-dispatch framework*. The BOT owner may assign *arbitrary* runtime policies for each BOT. These polices can depend on the system state, the BOT properties and state, the state of the different job replicas in the BOT, as well as various statistical properties of the resources. The policies can be adjusted during execution to accommodate unexpected changes in user requirements or system state.

Third, we enable resource-dependent prioritization policies to be specified for concurrently executing BOTs, so that multi-BOT scheduling algorithms can be used [21].

The GridBot system, which implements these policy-driven mechanisms, consists of a *work-dispatch server* and *grid execution clients* submitted to the grids by the *overlay constructor*. Our implementation is based on the BOINC server [13], developed as a part of the middleware for building community grids. Beyond its extensibility and proven scalability, BOINC is the de-facto standard middleware for building such grids. By integrating our mechanisms into BOINC, we make GridBot compatible with the standard BOINC execution clients, making it possible, in principle, to use over three million computers worldwide [1] where these clients are installed. Combined with the other clients dynamically deployed in grids to form the overlay, GridBot creates a unified scheduling framework for standard and community grids. To accommodate large number of resources we applied a number of optimizations for greatly increased scalability.

We envision the GridBot system to be employed by workflow engines, such as Pegasus [19] and DAGman [2]. However, our original motivation was to supply the growing computing demands of the Superlink project [33]. The Superlink-online Web portal [8] provides computing services for geneticists worldwide, enabling data analyses for detection of disease-provoking genetic mutations. The computational problem is exponential in the size of the data; hence the computing demands range from a few seconds to hundreds of CPU years. The data is submitted via a simple Web-based interface, and the analysis is automatically parallelized into a BOT with jobs of the desired granularity. The BOT sizes can range from a few jobs to millions, depending on the complexity of the input.

GridBot serves as a computing platform for running BOTs from Superlink-online, and is deployed in a pre-production setting. It currently utilizes resources in the OSG, EGEE, the UW Madison Condor Pool, the Technion campus grid, a number of local clusters, and the Superlink@Technion community grid [9]. During three months, about 25,000 computers worldwide have participated in the computations, with 4,000 from EGEE, 1,200 from Madison, 3,500 from OSG, and the rest from about 5,000 volunteers from 115 countries. GridBot's effective throughput roughly equalled that of a dedicated cluster of 8,000 cores, with theoretical peak throughput of 12 TFLOPs. Over 9 million jobs from about 500 real BOTs were executed, ranging from hundreds to millions jobs per BOT, requiring minutes to hours of CPU time for each job. The total effective CPU power consumed in three months equals 250 CPU years. (Due to the on-demand nature of the workload originating from the Superlink-online Web portal, there were also periods of idle time.) The current GridBot statistics are gathered via an extensive runtime monitoring infrastructure and are available online [6].

In our experiments we demonstrate the flexibility, efficiency and scalability of GridBot for running various real-life BOTs. We also evaluate common replication and scheduling policies on a scale which to the best of our knowledge has never been shown before.

Related work. From the onset of cluster and grid computing research, a number of systems have been developed for execution of BOT-type workloads using application-level scheduling (APST [17], Nimrod-G [12], Condor Master-Worker [20] among others). The recent works reemphasized the importance of overlay computing concepts (also termed multi-level scheduling) [3,22,29,35,37]. However, the existing systems do not provide BOT-specific execution mechanisms, leaving their implementation to the application. Nor can they utilize community grids or grids with strict firewall policies. Our approach is to enable the execution of BOTs in compound non-dedicated environments by making the BOT a first-class citizen at the work-dispatch level, thus removing the burden from the application, while allowing for the *application-specific policy* to be specified.

Condor glidein technology [3, 36] is the closest to GridBot in terms of its overlay computing and policy specification mechanisms [30]. However it currently lacks BOT-specific functionality in general and replication in particular. Furthermore, private networks and strict firewall policies pose significant obstacles to the use of glideins in standard and community grids. Yet, the success of Condor encouraged us to use classads as the policy language.

Falkon [29] achieves remarkable scalability and work-dispatch efficiency, but to the best of our knowledge it does not allow any parameterized policies to be specified.

Workflow engines, such as Swift [39], DAGman [2], Pegasus [19] and Nimrod-K [11], provide a convenient way to compose multiple BOTs or jobs into a composite parallel application. All of them allow execution over regular batch or overlay-computing systems, but do not expose the replication policy to the user.

The idea of replicating jobs in failure-prone environments was investigated from both theoretical [26] and practical perspectives [10, 14, 18, 23, 38]. These papers propose the algorithms for replication and resource selection to reduce BOT turnaround.

These works motivated the design of our replication and scheduling mechanisms and served as examples of policies to be enabled by GridBot.

Bundling of multiple jobs was suggested in the context of Pegasus [34] and Falkon [29].

Scheduling heuristics for multi-BOT scheduling were investigated by Iosup et al. [21] and Anglano et al [15], and served as a motivating example for our ranking policy mechanism.

Integration of different types of grids, including community grids, was also discussed by Cappello et al [16], and further developed by EDGeS [4] project. These works mostly focus on the system infrastructure, as opposed to the user-centric mechanisms of GridBot.

2. BACKGROUND

The term *grid* refers to a distributed *computing* environment with opportunistic *best-effort*, preemptive resource allocation policies. Namely, the jobs can be preempted by the resource manager at any moment, and neither the amount of available resources nor the time it takes to acquire them is bounded. In contrast, a *dedicated cluster* (or *cluster*, for short) is a computing environment with preemption-free allocation policy and short queuing times.

We further categorize grids into *collaborative* and *community* grids. Collaborative grids are formed by a federation of resources from different organizations, shared among the participants according to some agreed policy, e.g., *fair-share*. Community grids consist of the home computers of enthusiasts who donate them to one or several scientific projects.

The term *bag-of-tasks* (BOT) refers to a parallel computation comprised of independent jobs. Successful termination of all jobs is necessary for termination of the BOT.

Overlay computing is a technique for mitigating the long waiting times in grid queues whereby special execution clients are submitted to the grids instead of real jobs. When invoked on the grid resource, such a client fetches the jobs directly from the user-supplied work-dispatch server, thus bypassing the grid queues.

2.1 Resource management

Collaborative grids are built as a federation of clusters (not necessarily dedicated, in our terms). Each cluster is managed by the local batch queuing system, which along with the resource allocation policy for its local users, also obeys the global grid-wide user policies. In the following we focus on large-scale collaborative grids such as EGEE [5] and OSG [7]. The internal structure and the resources of each cluster in a grid are hidden behind the gateway node, which is used for job submission and monitoring. The compute nodes often reside on a private network or behind a firewall and local login to the nodes or the gateway is not permitted. The grid users submit jobs directly to the gateway or via Resource Brokers (as in EGEE).

Community grids rely on home computers around the world. They have been popularized by Berkeley Open Infrastructure for Network Computing (*BOINC*) [13] used for establishing community computing grids for specific scientific needs. Such a grid is managed by a single *work-dispatch server*, which distributes the jobs among the BOINC *clients*. The BOINC client, installed on a volunteer's computer, may connect to multiple work-dispatch servers, effectively sharing the machine between multiple community grids. This crucial feature makes the idea of establishing community computing grids particularly appealing. Indeed, in theory, over three million participating computers can be accessed. The only challenge, which is surprisingly much more complicated, is to motivate their owners to join the newly established grid.

2.2 Grid characteristics

In this section we analyze the properties of a number of collaborative and community grids in order to quantify the parameters affecting BOT performance. These will determine the realistic assumptions we can make while designing our solution.

In Figure 1(a) we present the history of the number of available resources as observed by a user having the steady demand of 1000 jobs, measured during one week in OSG and the UW Madison Condor pool. This short snapshot demonstrates typical behavior observed in opportunistic grids, and highlights the difficulty in providing short-term predictions of the resource availability. Observe the sharp changes (sometimes of an order of magnitude) in the number of allocated resources over short time periods. The growth in the number of resources during the 21st-22nd of February is, however, expected, as it coincides with the weekend. This variability prompts a design that *does not rely on static estimates* of the number of available resources.

Figure 1(b) shows the distribution of queuing times of a random sample of about 45,000 jobs invoked in the UW Madison pool and 12,000 jobs in the OSG, measured from the moment the job enters the batch queue until it starts execution. The measurements were performed in the steady state with 100 running jobs. Termination of one job triggered submission of a new one. Observe the variations in queuing times, which range from a few seconds to a few hours. Similar findings were reported in [27] for EGEE. These results unequivocally show that *overlay computing* is an absolute must for obtaining short turnaround time.

Figure 1(c) summarizes the failure rate of jobs 20 to 60 minutes long, measured during one month of operation in OSG, EGEE, UW Madison, the Technion cluster (100 cores) and the community grid Superlink@Technion with ~15,000 CPUs. Note that all the jobs executed in collaborative grids experience quite high failure rate due to preemptions, whereas failures due to hardware or software misconfiguration are rare. The community grids, however, have a low preemption rate and frequent hardware and software failures. Thus, any solution for BOT execution will have to *be optimized to handle job failures.*

3. GRIDBOT ARCHITECTURE

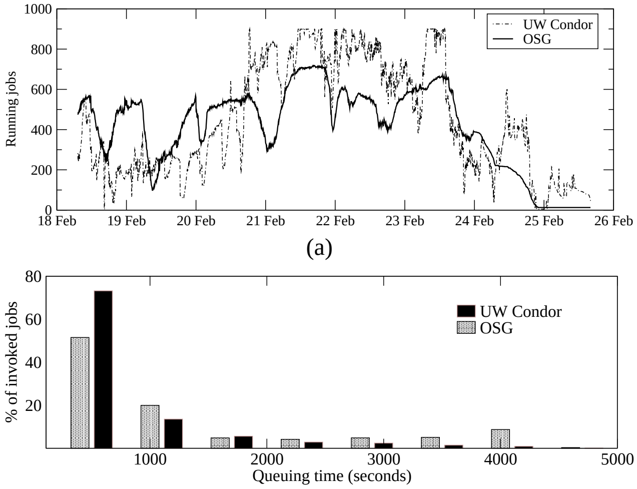
The GridBot architecture is depicted in Figure 2. It is logically divided into *work-dispatch* logic and the grid *overlay constructor*.

Execution clients in the overlay follow the pull model, whereby they initiate the connection to the server to fetch new jobs, but the server is not allowed to initiate the connection to the clients. We target the case where the traffic initiated from the public network to the clients is entirely disallowed. However, we assume that they can initiate connection to a single port of at least *one* host in the public space. This assumption holds in the majority of grid environments with which we had a chance to work.

The overlay constructor is responsible for submitting new execution clients into the grids whenever there are jobs in the job queue. It determines the number of clients to be submitted to each grid and issues the resource requests to one or more *submitters*. Note, however, that there are also static (as opposed to dynamically deployed via the overlay) clients which originate in a community grid. They are entirely under the control of the resource owners and contact the server at their will.

4. WORK-DISPATCH LOGIC

The work-dispatch logic comprises two interdependent components: the generic mechanisms for matching, prioritization, bundling, deadline and replication; and the policy evaluation mod-



(a)

(b)

Grid	#Jobs	Preempted (%)	Failed (%)
UW Madison	96938	20	1%
OSG	60648	10	1%
EGEE	16437	7	2%
Technion	42411	2	0.2%
Superlink @Technion	241364	(♣)0.2	13%

(♣) Job is counted as preempted if no results were returned before the deadline

(c)

Figure 1: (a) Resource availability. (b) Queuing time distribution. (c) Failure rate in different grids.

ule for enforcing the user-specified policies controlling these mechanisms.

As in Condor, we use classads for the policy specification. *Classified advertisements (classads)* [30] is a generic language for expressing and evaluating properties. A classad is a schema-less list of name-value attributes. It can be logically divided into a set of descriptive attributes having constant values, as in XML, and functional attributes specifying an arbitrary expression for computing their actual value in runtime. These expressions may include constants, references to other attributes, calls to numerous built-in functions, or nested classads. A classad interpreter enables efficient dynamic evaluation of the functional attributes at runtime, which, coupled with the schema-less nature of the language, opens unlimited possibilities for policy specification.

4.1 Classads in GridBot

Every system entity is described as a classad. Here we detail only the most important attributes in each classad, but in practice there are more of them, and new ones can be added.

The host classad contains the static and dynamic properties, some of which are reported by the host, such as number and type of CPUs, host owner name, the performance estimates and the number of currently running jobs on this host; and others maintained by the work-dispatch server and include long-term statistics, such as the job failure rate, the average turnaround time of jobs on that host, and the amount of CPU time used *recently* for producing error-free results.

The job classad for a non-replicated job has a small set of properties, such as job invocation parameters. However, if there are other running replicas of that job, the classad will be *dynamically extended* by the work-dispatch mechanism to include the *host* classad for each such replica. Hence, the scheduling and replication policies can refer not only to the current instance of the job, but to all the hosts executing the other replicas.

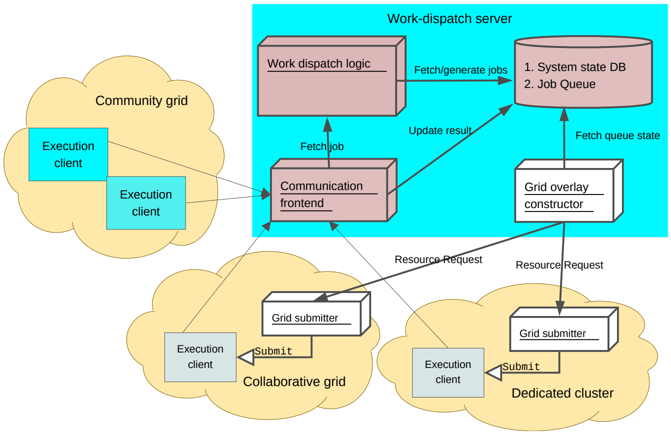


Figure 2: GridBot high level architecture

The BOT classad contains the number of incomplete jobs per BOT, and, most importantly, the *Tail* attribute, used to monitor the execution phase of the BOT. *Tail* is dynamically updated by the work-dispatch logic when the transition between the high-throughput and the tail phase occurs. Note that if *Tail* is used in some policy, the work-dispatch logic affects *its own* behavior at runtime. We elaborate on the tail phase detection in the implementation section.

The queue classad publishes the number of BOTs in the queue, allowing for the policies to refer the current queue load.

All the functional attributes, expressing the policies, are placed in the BOT classad, and shared among all the jobs of the BOT. They include *JobRequirements, ReplicationRequirements, Rank, Concurrency and Deadline*, and will be discussed later.

Figure 3 presents an example of a compound classad comprising BOT, Queue and Job classads. Observe that the Job classad also contains the classads of the hosts executing its replicas. The meaning of the policies (in bold) is explained below.

4.2 Policy-driven work-dispatch algorithm

The work-dispatch mechanism comprises the scheduling and replication phases, described in Algorithm 1 and 2 respectively.

The scheduling phase is invoked upon every job request. First, the host, queue and BOT classads are instantiated. Then, the job queue is traversed, for each job its classad instantiated, and all the policies evaluated given the specific values of job, BOT, host and queue attributes. The goal of the traversal is to find a *candidate set* of jobs, J, for which *JobRequirements* evaluate to true. Among the jobs in the candidate set, those having the highest *Rank* are selected. The number of jobs assigned to a host at any moment is determined by the value of the *Concurrency* attribute. Before sending the jobs to the host, the $deadline$ parameter for each job is assigned the value of the *Deadline* attribute.

The ability to assign multiple jobs per host allows pipelining, or bundling, used to reduce the per-job invocation overhead for shorter jobs. In the multi-BOT case the use of higher *Concurrency* by the BOT with lower *Rank* may lead to a violation of the prioritization policy. Hence, the value of the *Concurrency* attribute of the highest priority BOT is enforced.

The replication phase is executed by periodically traversing all *running* jobs. It is regulated by two policies: the job $deadline$ mentioned above, whose expiration signifies that the remote re-

```
[ Job= [ Name=”job1”;
         Executable=”/bin/hostname”;
         NumberOfReplicas=2;
         Replica1= [ Name=”job1_1”;
                     Host=[ Name=”is3.myhost”;
                            SentTime=242525;
                            ErrorRate=0.08;]
                   ];
         Replica2= [ Name=”job1_2”;
                     Host=[ Name=”is2.myhost”;
                            SentTime=242525;
                            ErrorRate=0.08;]
                   ];
      ];
  BOT= [JobRequirements = !Tail?  True:
              regexp(Host.Name,/*myhost*/)
              && Host.ErrorRate<0.1;
        Rank= !Tail?1:  Host.JobsToday;
        ReplicationRequirements=
          (NumberOfReplicas<3&&
          Job.Replica1.Host.ErrorRate>0.1);
        Concurrency=2*Host.NumCpus;
        Deadline=Concurrency*2000;
        JobsLeft=10;
        JobsDone=5;
        Tail=true;
      ];
  Queue= [BOTsInQueue=1;]
]
```

Figure 3: Example of a typical GridBot classad

Algorithm 1 Scheduling phase: upon job request from host h

```
Instantiate classad for $h$, BOT and queue
Foreach job $j$ in the job queue
  Instantiate classad for $j$
  Evaluate Concurrency$_j$, Deadline$_j$, JobRequirements$_j$ and Rank$_j$
  If JobRequirements$_j$ = true
    Add $j$ to candidate set J
End
Order the jobs in J by Rank$_j$
Concurrency←Concurrency of a job with maximum Rank
Foreach job $j \in$ J
  Concurrency←min(Concurrency, Concurrency$_j$)
  If Concurrency < assigned+#running jobs on $h$
    $deadline_j$ ←Deadline$_j$
    Assign job $j$ to host $h$
    assigned ← assigned+1
  End
End
```

source failed and the job should be restarted, and *ReplicationRequirements*, used to speed up the computations toward the end of BOT execution. The *ReplicationRequirements* attribute is evaluated only when the number of idle jobs that belong to the specific BOT becomes too low. Without this constraint, the replicated and not yet replicated jobs would contend for the resources, leading to the throughput degradation.

While both the *Deadline* and *ReplicationRequirements* policies control replication, they serve two completely different goals. The replication of jobs with an expired deadline is necessary in pull-based architectures where the client might not report its failure to the work-dispatch server. The deadline expiration ensures that any job executed by a faulty client will eventually be completed. In contrast, the *ReplicationRequirements* aim at reducing BOT turnaround time by increasing the likelihood of successful job termination.

Algorithm 2 Replication phase: once in replication cycle

```
Foreach running job $j$
/*Replication for expired Deadline*/
    Check the execution time $t$ of $j$
    If $t > deadline_j$
      Create new replica $j'$ and enqueue
      Mark $j$ as failed
      continue
    End
/*Replication for speculative execution*/
  If few unsent jobs of that BOT in the queue
    Find all replicas of $j$ and their respective executing hosts
    Instantiate classad for $j$
    If ReplicationRequirements = true
      Create new replica $j'$ and enqueue
  End
End
```

Several examples of possible policies are presented in Figure 3. The matching policy defined by the *JobRequirements* attribute allows for execution of a job on any host if the BOT is not in the tail phase, otherwise restricting it to those hosts having the string *myhost* in their names and low error rate. The *Rank* expression assigns higher relative priority to the jobs of this BOT on hosts which recently produced successful results, but this prioritization will be applied only in the tail phase. The *ReplicationRequirements* policy allows replication only if there are less than three replicas and the first one is running on a host with a high failure rate. The *Concurrency* expression allows a host to prefetch no more than two jobs for each CPU core. The *Deadline* attribute assigns the job $deadline$ parameter in accordance with the actual number of jobs sent to the host, and indirectly depends on the host properties in this case.

5. GRID OVERLAY

The overlay of execution clients is automatically established in the grids in response to the changing resource demand.

The grid overlay constructor distributes the client invocation requests between different grids under the following constraints:

1. Each grid restricts the number of concurrently executing or enqueued jobs

2. A grid job must not stay idle on the execution host, as happens when the execution client cannot receive new jobs from the work-dispatch server

The second constraint is particularly difficult to satisfy when the BOT *JobRequirements* policy prevents execution of jobs on hosts with specific properties, e.g., a policy which excludes the hosts with a high failure rate. Clearly, this information is inaccessible to grid submitters as it is not maintained by the native grid resource managers. Even if it were imported from the work-dispatch server, large scale grids typically disable fine-grained selection of individual hosts.

Our solution is based on two complementary techniques. First, the running client automatically commits suicide if it fails to obtain new jobs from the server or if it detects low CPU utilization by the running job. Second, we allow coarse-grained selection of grids via the BOT *GridPolicy* attribute. This attribute is evaluated by the overlay constructor in the context of grid classads published by the grid submitters. Once the set of suitable grids is determined, the problem becomes a variation of the classic *bipartite graph maximum matching* problem, where multiple BOTs must be matched to multiple grids subject to the constraints on the number of available resources in each grid and the resource demand of each BOT. We omit the details for lack of space.

6. IMPLEMENTATION

We implemented the work dispatch algorithm and integrated it into the existing BOINC server. We begin with a brief description of the original BOINC work-dispatch logic and then explain our own implementation.

6.1 BOINC

BOINC uses standard HTTP protocol for communication between the execution clients and the work-dispatch server. The server is based on the out-of-the-box Apache Web server. Data transfers are performed by the Web server, whereas the control flow is handed over to the custom backend.

The server *does not* maintain an open TCP connection with the clients during the remote execution of a job. Rather, clients immediately disconnect after fetching new jobs or reporting results. This design allows for impressive scalability with respect to the number of concurrently executing clients, but results in a delay in client failure detection until the $deadline$ expiration.

The server comprises several modules, in particular the *scheduler* and *feeder* , which implement the work-dispatch logic. The *scheduler* handles work requests from clients. This is a latency-critical component whose performance directly affects the system throughput. Thus, in order to hide the latency of accessing the job queue in the database, the *feeder* pre-fetches the jobs from the database and makes them available to the scheduler via a *shared-memory scheduling buffer.* The feeder is responsible for keeping this buffer full as long as there are jobs in the queue.

6.2 Integrating work-dispatch policies

Scheduling phase (Algorithm 1) cannot be implemented “as is”, because it requires the policies to be evaluated on *all the jobs in the queue* upon *each job request* . The size of the queue can easily reach a few million, rendering the policy evaluation infeasible.

One natural approximation of the algorithm would be to apply it to a random sample of the jobs in the queue. Yet, naïve uniform sampling is not applicable as the BOTs in the queue may have different number of jobs each, hence the larger ones would be over-represented in the sample. Instead, we apply the algorithm on the sample which includes the jobs of *all* enqueued BOTs. Hence, if there are n BOTs in the queue, we reserve at least $1/n$-th of the scheduling buffer capacity per BOT. To fill the relevant segment of the buffer, the feeder fetches the jobs of the BOT from the queue, redistributing the remaining buffer space among the other BOTs.

The jobs remain in the scheduling buffer until they are matched, or until their scheduling time-to-live timer expires. This timer prevents buffer congestion by the jobs with too restrictive *JobRequirements*. The jobs whose time-to-live timer has expired, are removed from the buffer and returned back to the queue for another scheduling attempt.

Replication phase (Algorithm 2) requires continuous monitoring of the deadline expiration of all running jobs. In practice, the expired jobs are detected via efficient database query without exhaustive traversal.

The evaluation of the *ReplicationRequirements*, however, cannot be offloaded to the database, as it is hard (if at all possible), to map the respective classad expression to the general database query. However, the algorithm evaluates the *ReplicationRequirements* attribute only when there are not enough enqueued jobs of the respective BOT, hence avoiding the overhead during the high-throughput phase. Furthermore, the feeder selects the candidates for replication via weighted sampling, where the weight is reverse proportional to the number of existing replicas of a job, to first replicate all the jobs having fewer replicas. We also restrict the maximum number of replicas per job to avoid unlimited replication.

6.3 Tail phase detection

We consider a BOT to be in the tail phase when the number of its jobs in the queue drops below a certain threshold, usually about the size of the scheduling buffer. Once this condition is satisfied, the feeder updates the *Tail* attribute in the BOT’s classad, making this information available to the work-dispatch logic. The advantage of such tail detection heuristics is that it does not require to estimate the number of the available resources, which cannot be done reliably.

The new jobs created as a result of replication (or job failure) may fill the queue again, causing the *Tail* attribute to turn back to false. Such fluctuations are sometimes undesirable and can be disabled, in particular when the *Tail* attribute is used to tighten the constraints on the scheduling policies in the tail phase, e.g., by allowing execution on more reliable hosts. On the other hand, the *Tail* attribute can be used for automatic adjustment of the replication rate if it becomes too high.

6.4 Scalability optimizations

System scalability depends mainly on scheduler’s ability to quickly choose the set of the jobs having the highest *Rank* for the requesting host. Since the *Rank* depends on the host parameters, no advanced indexing is applicable, hence only exhaustive traversal over all the jobs in the scheduling buffer will allow the precise actuation of the ranking policy. The *Concurrency* attribute further complicates the problem, as the number of jobs to be sent to a host depends on the host properties. The option of reducing the scheduling buffer is unacceptable, as it must be large enough to allow representation of all the enqueued BOTs.

Our optimization is based on the observation that the jobs of a single BOT are almost always *identical* from the scheduling perspective. Indeed, the policies are specified at the BOT level as all the jobs pertaining to the same BOT are assumed to share the same resource requirements. However, for the jobs with multiple running replicas, this similarity no longer exists. The scheduling policy may differentiate between jobs having multiple running replicas by considering the properties of the hosts where these replicas are being executed. One example is when the policy disallows invocation of multiple replicas of the same job in the same grid, in order to distribute the risk.

Applying the above optimization reduces the scheduling complexity from $O($#jobs in scheduling buffer$)$ to $O($#BOTs in scheduling buffer$)$. Also, it enables the rejection of unmatched hosts upfront, which is very important when community grids are part of the infrastructure, as the clients cannot be prevented from contacting the server. This optimization significantly increases the scalability while still covering most of the important scheduling policies, as will be shown in the experiments.

6.5 Execution clients

The overlay is formed by BOINC clients submitted to the grid by the grid submitters. However, a few modifications to the existing clients were required, involving some non-trivial changes to allow proper error handling in grids.

We focus on the following types of failures:

1. Failures due to faulty resources which continuously produce errors immediately after starting the job – “black holes”

2. Network failures or failures due to the excessive server load

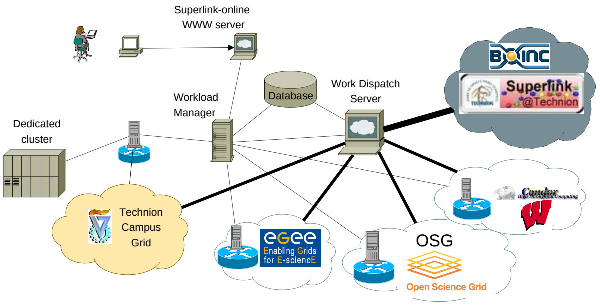


Figure 4: Deployment of GridBot for Superlink-online system

Black hole elimination requires the client statistics to be stored on the server. The client generates a unique random identifier when first contacting the server, and uses it in all future communications. This identifier is supposed to be stored persistently in the machine where the client is installed. In grids, however, the client state is wiped from the execution machine after preemption, which effectively results in the loss of the server-side statistics. We solved the problem by generating a consistent identifier using the host MAC address.

Network failures are frequent in wide area networks, and BOINC clients automatically retry the failed transaction with exponential back-off. In grids, however, the back-off eventually leads to automatic self-termination of the client to avoid grid resource idling. Our attempt to shorten the back-off solved this particular problem, but resulted in an exceedingly high network traffic (which in fact was classified as a DDoS attack) when the real cause of the failure was the server overload and not the network outage. Hence, for large scale deployments, the exponential backoff must be in place even at the expense of efficiency.

7. RESULTS

The development of GridBot was primarily motivated by the Superlink-online system, which performs statistical analysis of genetic data for detecting defective disease-provoking genes in humans and animals. It is accessible via a simple Web interface, and is used by geneticists worldwide. Since 2006, over 18,000 analyses have been performed by the system. The analysis is automatically parallelized and transformed into a BOT with jobs of the required granularity [33]. The computational demands vary significantly among different inputs, ranging from a few CPU seconds to hundreds of CPU years.

The experiments described in this section serve three main goals: to compare GridBot with the other alternatives for running BOTs; to evaluate its scalability as a function of the number of jobs and BOTs in the queue, and the job request rate; and to demonstrate the flexibility of the policy specification as well as the impact of different scheduling and replication policies in a large-scale multi-grid environment.

We performed *all* the experiments using real data from the runs that were previously invoked via Superlink-online. Namely, we used the genetic data previously submitted by the users of Superlink-online, effectively re-executing the analysis, in some cases several times to obtain statistically valid results.

The GridBot deployment used for these experiments is shown in Figure 4. Nor the exclusive access to the resources was granted, neither any of them were reserved during the experiments. Instead, all the grid resources were allocated on purely opportunistic basis via local grid management infrastructures.

The current deployment features the fail-over dedicated cluster, in addition to the grids. Jobs that fail repeatedly in the grids are automatically transferred to this cluster to be invoked in the controlled environment.

Naive execution via BOINC overlay. We executed a medium-sized BOT using resources in all available grids. For this experiment we replaced the policy-driven work-dispatch server with the *unmodified* BOINC server. The rest of the GridBot system was left unchanged. The experiment was repeated five times and the best run selected. The graph in Figure 5 shows the distribution of the number of incomplete jobs over time. Observe the high consumption in the throughput phase and the slow tail phase.

The graph also demonstrates how the *Deadline* parameter affects the job execution. *Deadline* was set to three days for all jobs. This was the minimum acceptable value for the volunteers in Superlink@Technion grid. The reason for such a long deadline is in the structure of community grids in general, most of which assign deadlines of several weeks. Since a single client is connected to many such grids, those with shorter deadlines (less than three days) effectively require their jobs to be executed immediately, thus postponing the jobs of the other grids. This is considered selfish and leads to contributor migration and a bad project reputation, which together result in a significant decrease in throughput.

Observe that some of the results were returned more than 30 hours after they were sent for execution. In general, we found that the ratio between execution time and turnaround time for the jobs in the community grid varies between 0.01 to 1, with the average at 0.3 (as opposed to 1 for collaborative grids).

The execution of the same BOT by GridBot using the same set of resources required only 8 hours versus 280 by the naive execution, without violating the minimum deadline constraint for community grids.

GridBot versus Condor. We compared the turnaround time of a BOT executed via GridBot under the policy to route jobs only to the UW Madison Condor pool, with the turnaround time of that BOT executed directly via Condor in the same pool. Comparison with Condor is particularly interesting since GridBot implements a matching mechanism similar to that of the Condor work-dispatch daemon. This setup gives an advantage to Condor because its work-dispatch daemon is located close to the execution machines in Madison, whereas the GridBot server resides in Israel.

To put GridBot under high load, we ran a BOT of 3000 short jobs ranging from 30 seconds to 5 minutes. GridBot was configured with a 10 minute *Deadline*. The replication policy allowed replication of a job if the failure rate of the running host was above 10%. The BOT was executed five times in each system.

The average turnaround time in GridBot was 53+/-10 minutes, versus 170+/-41 minutes in Condor, with GridBot faster, on average, by a factor of 3. Less than 1% of the jobs were replicated. This result proves that the execution via GridBot does not introduce any overhead as compared to Condor, and in this case (small BOTs with short jobs) is even more efficient.

High throughput run. We invoked a BOT with 2.2 million jobs ranging from 20 to 40 minutes. The BOT was completed in 15 days. The accumulated CPU time (sum of the times measured locally by each execution client, hence excluding communications) used by the system for this run is 115 CPU years. The effective throughput is equivalent to that of a dedicated cluster of 2,300 CPUs. The BOT execution involved all nine available clusters and grids. The contribution by the five main grids is summarized in Figure 6(c).

Figure 6(a) depicts the change in the number of incomplete jobs during the run. The almost linear form of the graph suggests

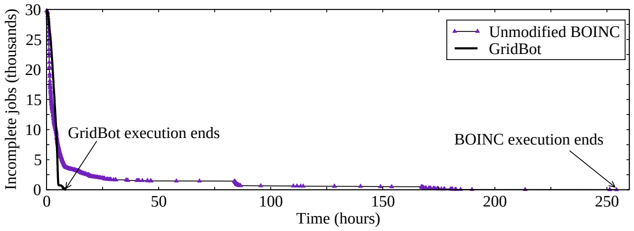


Figure 5: Naïve execution of BOT in multi-grid

that GridBot consistently managed to recruit a large number of resources despite the high volatility of grid resources.

Figure 6(b), which is a snapshot of the online GridBot Web console [6], presents the effective throughput of the system during the last week of this run. In this chart, *Community*, *OSG*, *EGEE*, *Madison-condor*, *T-condor* signify the Superlink@Technion community grid, the Open Science Grid, the EGEE grid, the large Condor pool in UW Madison, and the Technion dedicated Condor pool, while all the rest represent different clusters and groups of desktop machines in the Technion campus.

In a non-dedicated environment, the number of concurrently executing CPUs cannot be used to estimate the throughput because of the job failures. To obtain a more realistic estimate, we periodically sampled the running time of 1000 recently finished jobs, and multiplied their average by the number of jobs consumed since the last sample. Provided that all the jobs have similar computing demand, the obtained estimate yields the number of CPUs in an imaginary fully dedicated cluster. The capacity of each CPU in this imaginary cluster equals the average over the capacities of all CPUs contributing to the computation (currently 1.4 GFLOPs per CPU according to our data).

Observe the spike in the throughput on 24/02 in Figure 6(b). There was a network failure in the Technion, and the spike occurred immediately after the network was restored. This is a consequence of the prefetching of jobs by the clients. The clients continued to execute the already prefeched jobs despite their inability to access the server and uploaded them when the connectivity was restored.

The tail phase began in the evening of 24/02, when there were about 60,000 incomplete jobs. All these jobs had already been sent for execution, and the queue was empty. This also coincides with the throughput drop in the grids, since the overlay constructor recognized the decreased resource demand.

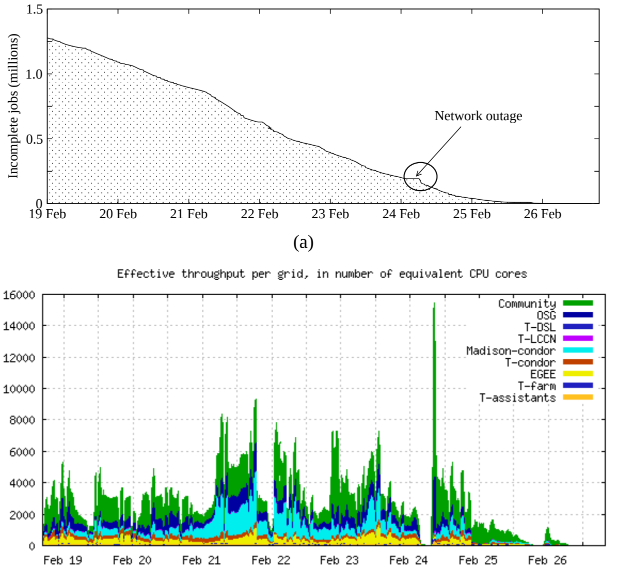
This run demonstrates the unique capabilities of GridBot to employ multiple grids under a unified scheduling framework.

Influence of the scalability optimization. We evaluated how the scalability optimization (Section 6.4) affected the throughput.

GridBot was configured with a scheduling buffer of 1500 jobs and invoked with and without the optimization. We used a BOT of 29,000 jobs, 10-15 minutes each. For each configuration we ran the BOT twice: the first time over all collaborative grids, and the second time over all resources including the community grid.

Both configurations performed equally over the collaborative grids, with a peak job dispatch rate of about 3 jobs per second. However, the run over all grids failed in the non-optimized mode. The increased job request rate was beyond the server's capabilities and led to the buildup of pending requests. The current implementation failed to handle this overload properly, and at some point started to drop all the requests without serving them.

While the optimization was crucial to allow scaling to more resources, even the non-optimized mode was scalable enough to enable the run over all collaborative grids.



(a)

(b)

Grid	Throughput (%)	#Hosts
UW Madison	20	1113
OSG	10	1601
EGEE	16	2895
Technion	5	121
Superlink@Technion	47	4067

(c)

Figure 6: High throughput run statistics: (a) The number of incomplete jobs over time. (b) Throughput across different grids over time. (c) Aggregate statistics per grid.

Scalability benchmarks. We evaluated the scalability of GridBot across three dimensions: number of jobs in the queue, number of BOTs in the queue, and number of job requests per second. GridBot was configured in the optimized mode in all the experiments.

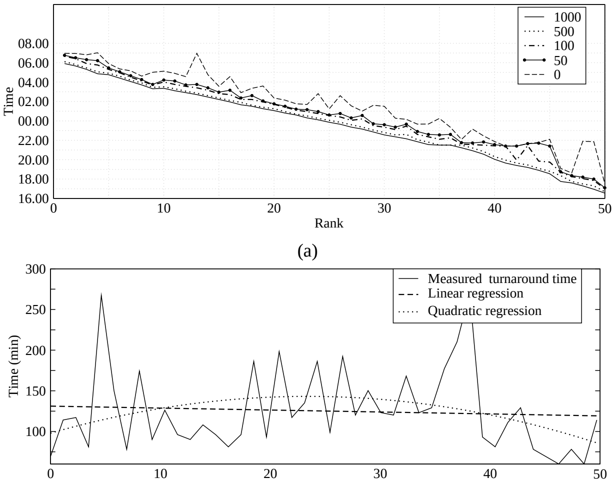
We observed no change in performance even with several million of enqueued jobs, as was also demonstrated by the high throughput run above. This was expected, as the work-dispatch logic does not depend on this parameter. The other two parameters were evaluated by imposing high load in a real large-scale setup.

Number of BOTs in the queue. We submitted 50 BOTs at once, each with 1200 jobs with an average running time of three minutes per job. This number of BOTs exceeds by a factor of five the peak load observed in the production Superlink-online system. We restricted the BOT execution to EGEE,OSG,UW Madison, and the Technion cluster. However, additional load was created by the hosts from Superlink@Technion, which cannot be prevented from contacting the server.

We assigned a unique *Rank* between 1 and 50 to each BOT to force the system to execute them one-by-one. Thus, we expect the higher priority BOTs to approach termination before the low-priority BOTs are started.

Figure 7(a) shows the results of the experiment. The statistics depicted in the graph were calculated as follows. The BOTs are ordered by *Rank* in an increasing order. For each BOT we recorded the time stamp when the number of incomplete jobs reached 1000, 500, 100, 50 and 0 (BOT terminated). One curve connects the respective timestamps of all the BOTs. The ordinate represents the time in the system.

Ideally, we expect the graph to decrease linearly from the low-



(a)

(b)

Figure 7: Scalability benchmark for the number of BOTs in the queue: (a) Precision of the ranking policy. (b) Distribution of the turnaround times.

est priority to the highest, indicating that the BOTs are executed according to their priorities.

Figure 7(a) demonstrates that the system behaves exactly as expected. According to the curve corresponding to the beginning of each BOT (1000 jobs left), none of the lower-ranked BOTs was started before those with higher priority. Observe also that the curve for 500 remaining jobs closely follows the one for 1000 jobs. Hence the bulk of computations for each BOT is performed according to the correct order. The deviation from this ideal behavior starts when the number of incomplete jobs drops below 100. This behavior is expected in the tail phase, where the execution progress becomes affected by random resource failures.

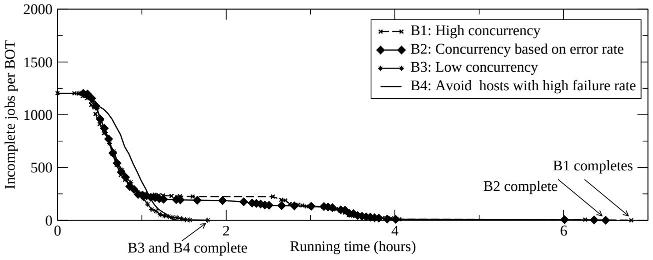
While *Rank* is constant in this experiment, the work-dispatch algorithm is not optimized for this case and evaluates the *Rank* attribute as usual. Thus, the correct execution in this experiment indicates correct execution with an arbitrary *Rank* expression.

Figure 7(b) shows the distribution of BOT turnaround times as a function of the number of BOTs in the queue. Clearly the performance is not affected by the larger number of enqueued BOTs.

Number of job requests. We invoked a single BOT with 42200 jobs, 10 to 50 minutes per job. The maximum job request rate during the execution of this BOT on all the available resources reached 4 requests per second. Since *Concurrency* was set to 5, which is reasonable for high-throughput runs with millions of jobs, the effective job dispatch rate reached 20 per second. The peak throughput was equivalent to that of a cluster of 3,700 dedicated cores.

Next we invoked the same BOT, but this time we split each original job into five. Hence this run involved 211,000 jobs of a few minutes each. As expected, the job request rate increased and reached 18 requests per second from different clients, leading to an effective work dispatch rate of 93 jobs per second. Despite the high load, we observed no degradation in the overall throughput.

Unfortunately, our attempt to push the limits even further did not succeed. The increase in network traffic and the rate of establishing new inbound connections to the server was classified by the central Technion firewall as a DDoS attack, and resulted in severe network degradation. Thus the network, rather than the work dispatch implementation, becomes the major bottleneck when scaling beyond 30-40 job requests per second.



(a)

	Concurrency	*JobRequirements*	**Turnaround**
B1	`10`	Grids	6.8h

B2	`Host.ErrorRate`
	`<0.01?10:1`
		Grids	6.5h

B3	`1`	Grids	1.8h
B4	`1`	Grids && good host reputation	1.4h

(b)

Figure 8: Influence of policies on turnaround time for small BOTs.

Influence of policies on turnaround time for small BOTs. Figure 8 presents the results of the execution of four copies of the same BOT with 1200 jobs of three minutes each. All BOTs were invoked one after the other. The policies used in this experiment appear in 8(b).

The scheduling policy of all BOTs allowed the use of the collaborative grids (OSG,EGEE and UW Madison) and the Technion cluster. B4 further restricted the execution to hosts having low failure rate and a recent history of successful job execution.

All BOTs had the same *Deadline*, and the same conservative replication policy, allowing a job to be replicated only once in (*Concurrency*×1000) seconds.

We see that for such small BOTs, the execution of jobs on potentially faulty hosts should be avoided (Figure 8(a)). We also observe that the increase of *Concurrency*, namely, the increase in the size of bundle of jobs assigned to the same host, results in a longer tail phase. This is caused by the increased failure rate due to the higher chances of such job to be preempted. Such a policy should thus be avoided toward the end of the run. Among these policies, B4 seems to behave best as it dynamically chooses the hosts with higher chances to complete the jobs. Such a policy, however, is likely to be beneficial ony for shorter BOTs because of the decreased throughput versus the more permissive ones.

Influence of replication policy in collaborative grids. We evaluated the influence of the replication and scheduling policies in collaborative grids alone. In each run we invoked a single BOT with 30,000 jobs, 10-15 minutes each. The results in the table below are averaged over five runs for each policy.

The *Permissive* replication policy allowed up to 5 replicas per job, whereas the *Restrictive* one allowed replication only if one of the replicas was running on an unreliable host (error rate higher than 1%, no recent successful results) or longer than 30 minutes. The *Permissive* scheduling policy allowed execution on all hosts. The *Restrictive* scheduling added the constraint to allow only reliable hosts during the tail phase. We measured the portion of jobs created during replication (Replicas column) and the portion of jobs whose results were discarded as there was already one result available (Waste column) out of the number of jobs in the run without replication.

We see that allowing unlimited replication is both wasteful and inefficient. Replicas of the same job compete for resources, which

	Replication	Scheduling	Replicas (%)	Waste (%)	Turnaround
P1	Permissive	Permissive	58	30	4.1h
P2	Restrictive	Restrictive	11	7	3.2h
P3	Permissive	Restrictive	73	57	4.2h
P4	Disabled	Restrictive	0	0	5.1h
P5	Disabled	Permissive	0	0	5.8h

Figure 9: Influence of replication policy in collaborative grids.

effectively slows down the execution. When the replication is disabled, the policy P4 which restricted the execution to the reliable resources toward the end of the run achieved better results.

These are only a few examples of the intuitively correct policies. But the true power of GridBot is in the ability to accommodate any user-specified runtime policies, which is impossible in any other system for BOT execution.

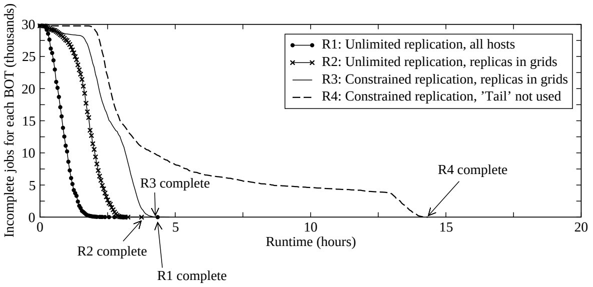
Replication with multiple BOTs and community grid. The experiment focuses on the influence of replication policies on the turnaround time when multiple BOTs are in the queue. In addition, we allowed the use of community grid resources. We invoked the same BOT as previously, but instead of submitting one BOT each time, BOTs were assigned different priorities and invoked simultaneously. The evaluated policies appear in Figure 10(b). The *Permissive* replication policy allowed up to 5 replicas for a job. The *Restrictive*$_1$ allowed replication once in 15 minutes, and only if at least one of the existing replicas is running on a host with high error rate. The *Restricitve*$_2$ was set to replicate only when the number of incomplete jobs dropped below 2000. The *Permissive* scheduling policy allowed execution on all machines. The *Restrictive*$_1$ added the constraint of disallowing community grid resources in the tail phase. The *Restrictive*$_2$ was similar to *Restrictive*$_1$, but the beginning of the tail phase was recognized via the static condition of the number of incomplete jobs being below 2000, rather than the dynamic *Tail* attribute.

Figure 10(a) shows the results of the run. We see that R1 has longer turnaround time despite the *Permissive* replication policy, which agrees with the results of the same policy in the previous experiment. R2 results in faster execution, and so does R3, despite the restrictive replication policy.

R4 proved particularly inefficient, due to the incorrect estimate of the expected beginning of the tail phase. Observe that the true tail phase began when there were about 10,000 incomplete jobs, in contrast with the 2000 jobs threshold estimated a priori. Furthermore, all the jobs which failed during the true tail phase were resubmitted to the community grid hosts, as all the collaborative grids were still occupied by R3. Once we changed the threshold from 2000 to 4000 jobs, it completed in 80 minutes.

Observe that the number of replicas generated in this experiment is larger than the number of replicas in the one involving only collaborative grids, even by the restrictive policies. This is because of the use of the community grid for such relatively short BOTs. From 30% to 70% of all jobs were sent to community grid resources, characterized by a higher turnaround time. Hence, even the restrictive replication policy allowed replication of such jobs. Most of the original jobs succeeded but the results were discarded.

We also see the influence of excessive replication when there are multiple BOTs in the queue. Observe that the time between R4 and R3 is shorter than that between R3 and R2, or R2 and R1. Observe also the knee in the graph of R3 when it reaches 13,000 jobs. The BOT tail was detected, and the new jobs were sent to the grids, which were already occupied by R2.



(a)

	Replication	Scheduling	Replicas (%)	Waste (%)
R1	Permissive	Permissive	188	105
R2	Permissive	Restrictive$_1$	129	75
R3	Restrictive$_1$	Restrictive$_1$	49	25
R4	Restrictive$_2$	Restrictive$_2$	35	21

(b)

Figure 10: Influence of replication policy in a mixture of community and collaborative grids and multiple BOTs.

Summary. The main goal of the extensive large-scale experiments performed in this work is to highlight the importance of the flexibility of policy specification. This is illustrated in the last three experiments, where the results of using similar policies in different settings emphasize the necessity of policy adjustment.

Moreover, the experiments show that some non-trivial policies yield better turnaround time, which is often the user's main concern. Namely, preferring reliable resources in the tail phase is worthwhile for both longer and shorter BOTs, whereas constraining the resources during the entire run reduces the effective throughput for the former. Static bundling of multiple jobs may increase BOT turnaround time due to the increased probability of job failures. Dynamic bundling based on the host failure rate achieves better results, though behaves significantly worse than no bundling at all if the bundle size is not tailored to the actual grid characteristics.

Our experiments also confirm that proper replication policy improves turnaround time but wastes resources. When the gap between the required and actual turnaround times of jobs invoked on grid resources is large, more replicas are created. This is the case for community grids. But replicating a job without considering the reputation of the machines already running other replicas results in less efficient resource use and even increases the turnaround time. Furthermore, allowing replication in the multi-BOT case efecitvely delays the other BOTs in the queue. However, while undesirable in multi-user or pay-as-you-use environment, these artifacts of replication are less of a concern if the timely completion of a certain BOT is required.

8. CONCLUSIONS

We presented GridBot, a policy-driven mechanism for efficient execution of BOTs on a platform unifying multiple grids of different types. We demonstrated the flexibility and scalability of GridBot by executing numerous real BOTs, with hundreds to millions of jobs each, on thousands of CPUs from multiple large-scale collaborative and community grids. None of the policies used in the experiments behaves equally well in all the settings. Hence, the ability of GridBot to accommodate an arbitrary dynamic policy is essential for efficient execution of BOTs under dynamic environmental conditions and changing user preferences.

We believe that production grid systems can benefit from adding BOT replication and scheduling mechanisms. For example, extending Condor with such mechanisms seems to fit naturally in the existing system design.

Lessons learned. GridBot reemphasizes the benefits of using overlay computing for executing jobs over grids. In addition to the performance benefits for the grid user, bypassing the grid queues significantly reduces the load on the grid gateways, thus increasing the overall system performance. However the overlay may also have negative effects on the grids. Working closely with grid administrators in the design of GridBot, we learned a number of important lessons in this regard.

The overlay execution client might occupy the remote resource forever if not terminated by the grid system. While having little effect in systems with preemptive scheduling, this behavior may lead to starvation of the other grid jobs in preemption-free environments, and to the underutilization of grid resources. Furthermore, the discrepancy between the information available to the overlay constructor layer and the one used by the scheduler may lead to the resources being occupied but not used. Another problem arises from the "infinite supply" of overlay execution clients, which are continuously being pushed to the grids to maintain the overlay instead of those clients which were terminated or evicted. It is thus essential to avoid queue overload on the grid gateways by limiting the number of submitted jobs. Furthermore, without active runtime monitoring of the load, even a single permanently failing resource may render the grid gateway unusable if the execution clients are submitted blindly and destined to failure. Finally, the open communication channel created by the overlay directly into the grid resources allows easy bypass of the grid security mechanisms. Not only does the security of the whole grid system become dependent on that of the overlay infrastructure, but the identity of the users running the jobs via the overlay is hidden from the grid accounting system.

Another important component of GridBot is job replication. However, as has been shown in the experiments, an incorrect replication policy may significantly increase the workload and decrease system efficiency. While GridBot implements mechanisms to avoid excessive replication, the impact of replication policies on the performance of the entire grid still have to be investigated.

Future work. Our recent [32] and future work focuses on the inclusion of GPU farms and GPU grids in our framework, as well as extending the overlay constructor to allow efficient use of pay-as-you-use cloud infrastructures in conjunction with other grids. In collaboration with the Tokyo Institute of Technology, GridBot is being enhanced to integrate the TSUBAME supercomputer into the same scheduling framework with other grids: this enhancement is facilitated by the flexible scheduling mechanisms presented in this paper.

9. ACKNOWLEDGMENTS

We are grateful to Prof. Miron Livny for many valuable discussions, for his strong support of the Superlink-online system and for allowing us to access the computing resources in OSG and UW Madison. We are also grateful to Dr. David Anderson who added a number of features to BOINC client to allow its execution in grid environments, and helped us understand the design of BOINC scheduler. We would like to thank all the administrators in OSG GLOW and EGEE Biomed virtual organizations, and in particular Dan Bradley from Condor team, for their patience and help in making GridBot grid-friendly. Finally we thank all the contributors of the Superlink@Technion community grid for allowing us to use their computers and for the educating feedbacks on the social aspects of community grids.

10. REFERENCES

[1] Community grids managed by BOINC. `http://http://boincstats.com`.

[2] Condor DAGman. `http://www.cs.wisc.edu/condor/dagman`.

[3] Condor Glidein. `http://www.cs.wisc.edu/condor/glidein`.

[4] EDGeS project. `http://www.edges-grid.eu/`.

[5] The enabling grids for e-science. `http://www.eu-egee.org`.

[6] GridBot monitoring. `http://cbl-boinc-server2.cs.technion.ac.il/superlinkattechnion/stripcharts.php`.

[7] The open science grid. `http://www.opensciencegrid.org`.

[8] Superlink-online genetic linkage analysis portal. `http://bioinfo.cs.technion.ac.il/superlink-online`.

[9] Superlink@Technion community grid. `http://cbl-boinc-server2.cs.technion.ac.il/superlinkattechnion`.

[10] J. H. Abawajy. Fault-tolerant scheduling policy for grid computing systems. In *IPDPS*, pages 238+, 2004.

[11] D. Abramson, C. Enticott, and I. Altinas. Nimrod/k: towards massively parallel dynamic grid workflows. In *SC '08*, pages 1–11, 2008.

[12] D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with Nimrod/G: Killer application for the global grid? In *IPDPS*, pages 520–528, 2000.

[13] D. P. Anderson, E. Korpela, and R. Walton. High-performance task distribution for volunteer computing. In *e-Science*, pages 196–203, 2005.

[14] C. Anglano, J. Brevik, M. Canonico, D. Nurmi, and R. Wolski. Fault-aware scheduling for bag-of-tasks applications on desktop grids. In *GRID*, pages 56–63, 2006.

[15] C. Anglano and M. Canonico. Scheduling algorithms for multiple bag-of-task applications on desktop grids: A knowledge-free approach. In *IPDPS*, pages 1–8, 2008.

[16] F. Cappello, S. Djilali, G. Fedak, T. Hérault, F. Magniette, V. Néri, and O. Lodygensky. Computing on large-scale distributed systems: Xtremweb architecture, programming models, security, tests and convergence with grid. *Future Generation Comp. Syst.*, 21(3):417–437, 2005.

[17] H. Casanova and F. Berman. Parameter sweeps on the grid with APST. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, chapter 26. 2003.

[18] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauve, F. A. B. Silva, C. O. Barros, C. Silveira, and C. Silveira. In *ICPP*, pages 407–416, 2003.

[19] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, V1(1):25–39, March 2003.

[20] J.-P. Goux, S. Kulkarni, J. Linderoth, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *HPDC*, pages 43–50, 2000.

[21] A. Iosup, O. O. Sonmez, S. Anoep, and D. H. J. Epema. The performance of bags-of-tasks in large-scale distributed systems. In *HPDC*, pages 97–108, 2008.

[22] G. Juve and E. Deelman. Resource provisioning options for large-scale scientific workflows. pages 608–613, Dec. 2008.

[23] D. Kondo. *Scheduling task parallel applications for rapid turnaround on desktop grids*. PhD thesis, 2005.

[24] D. Kondo, F. Araujo, P. Malecot, P. Domingues, L. M. Silva, G. Fedak, and F. Cappello. Characterizing result errors in Internet desktop grids. In *Euro-Par*, pages 361–371, 2007.

[25] D. Kondo, A. A. Chien, and H. Casanova. Resource management for rapid application turnaround on enterprise desktop grids. In *SC '04*, page 17, 2004.

[26] G. Koole and R. Righter. Resource allocation in grid computing. *J. Scheduling*, 11(3):163–173, 2008.

[27] D. Lingrand, J. Montagnat, and T. Glatard. Estimating the execution context for refining submission strategies on production grids. Technical Report I3S/RR-2007-22-FR, I3S laboratory, Sophia Antipolis, Nov. 2007.

[28] R. Medeiros, W. Cirne, F. V. Brasileiro, and J. P. Sauvé. Faults in

grids: Why are they so bad and what can be done about it? In *GRID*, pages 18–24, 2003.

[29] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a fast and light-weight task execution framework. In *SC '07*, pages 1–12, 2007.

[30] R. Raman. *Matchmaking frameworks for distributed resource management*. PhD thesis, 2000.

[31] M. Silberstein, D. Geiger, A. Schuster, and M. Livny. Scheduling mixed workloads in multi-grids: The grid execution hierarchy. In *HPDC*, pages 291–302, 2006.

[32] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens. Efficient computation of sum-products on GPUs through software-managed cache. In *ICS*, pages 309–318, June 2008.

[33] M. Silberstein, A. Tzemach, N. Dovgolevskiy, M. Fishelson, A. Schuster, and D. Geiger. On-line system for faster linkage analysis via parallel execution on thousands of personal computers. *American Journal of Human Genetics*, 78(6):922–935, 2006.

[34] G. Singh, M.-H. Su, K. Vahi, E. Deelman, B. Berriman, J. Good, D. S. Katz, and G. Mehta. Workflow task clustering for best effort systems with pegasus. In *MG '08*, pages 1–8, 2008.

[35] Y. suk Kee, C. Kesselman, D. Nurmi, and R. Wolski. Enabling personal clusters on demand for batch resources using commodity software. In *IPDPS*, pages 1–7, 2008.

[36] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[37] E. Walker, J. P. Gardner, V. Litvin, and E. L. Turner. Personal adaptive clusters as containers for scientific jobs. *Cluster Computing*, 10(3):339–350, 2007.

[38] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. pages 29–42, San Diego, CA, 12/2008 2008. USENIX Association.

[39] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Services 2007*, pages 199–206, 2007.