

# Efficient Allocation and Composition of Distributed Storage

Jimmy Secretan, Malachi Lawson and Ladislau Bölöni  
 School of Electrical Engineering and Computer Science  
 University of Central Florida  
 Orlando, FL 32816  
 jsecretan@ucf.edu, malachee@hotmail.com, lboloni@eecs.ucf.edu

## Abstract

In this paper we investigate the composition of cheap network storage resources to meet specific availability and capacity requirements. We show that the problem of finding the optimal composition for availability and price requirements can be reduced to the knapsack problem, and propose three techniques for efficiently finding approximate solutions. The first algorithm uses a dynamic programming approach to find mirrored storage resources for high availability requirements, and runs in the pseudo-polynomial  $O(n^2c)$  time where  $n$  is the number of sellers' resources to choose from and  $c$  is a capacity function of the requested and minimum availability. The second technique is a heuristic which finds resources to be agglomerated into a larger coherent resource, with complexity of  $O(n \log n)$ . The third technique finds a compromise between capacity and availability (which in our phrasing is a complex integer programming problem) using a genetic algorithm. The algorithms can be implemented on a broker that intermediates between buyers and sellers of storage resources. Finally, we show that a broker in an open storage market, using the combination of the three algorithms can more frequently meet user requests and lower the cost of requests that are met compared to a broker that simply matches single resources to requests.

## Index Terms

Grid Computing, Quality of Service, Distributed Storage, Grid Economics

## I. INTRODUCTION

There is an abundance of unused storage resources connected to the Internet. There are also a number of users who would benefit from the ability to rent network storage, provided that the required size and performance characteristics are met.

To make use of these idle resources, and to create opportunity for commercial providers to bring new storage services online, an open marketplace is needed to help connect buyers to sellers as well as to establish prices. This market approach to grid computing is emerging as a popular paradigm in the literature. A buyer in this economy would like to meet his availability and capacity requirements with the lowest cost solution possible. However, there might not be an exact match for the requirements of the users in the pool of available resources. To solve this problem, we propose a model in which the user assembles a resource of desired capacity and availability by combining resources which, individually, do not satisfy the user's requirements.

Note that a similar technique is used in local resources such as RAID arrays. The various RAID levels such as RAID0 (striping), RAID1 (mirroring), and so on are just different ways of combining storage resources to meet the storage, reliability, performance and cost demands. Our approach extends this model to a dynamic networked environment.

Our objective is to design a broker-based architecture for the efficient allocation of the resources. As the optimal composition problem has a non-polynomial complexity, we are interested in finding efficient approximate algorithms, with modest computational requirements.

This paper is organized as follows. In Section II, we discuss current work related to utilization of spare computational and storage resources, as well as the economics of trading and buying resources in a grid environment. In Section III, we introduce an algebraic and graphical notation to express composition of storage resources based on availability, capacity and cost. In Section IV, we use the notation to analyze the complexity involved in these resource optimization problems. In Section V, we describe a series of resource allocation experiments to test our algorithms, along with results and a discussion. Finally, in Section VI, we discuss future enhancements to our work.

## II. RELATED WORK

### A. Redundant and High Performance Distributed File Systems

With the popularity of cluster computing and network computing, many storage systems have been designed to offer high-performance and reliable storage by pooling together small and less-reliable commodity components.

In [1], an architecture called ClusterRAID is designed for high reliability and integrity data storage using nodes of a cluster. Redundancy information of an individual node's data is stored in such a way as to maximize reliability and availability, while minimizing network bandwidth. In the case of a node failure, the data can be reconstructed on any spare node. The use of Reed-Solomon algorithms allow the user to specify how many faults can be tolerated. A similar system, netRAID [2], uses a RAID3 style storage scheme, and is able to rebuild online in the event of a node failure.

The Cluster File System (CFS) [3], presents a solution for distributed video storage, aiming for high reliability, low cost and transparent use. The video streams are stripped across the nodes, following the RAID paradigm. The system has the option of using additional parity blocks (analogous to RAID5) to allow the system to recover from the failure of a single node. CFS is unique in that it is optimized for the delivery of sequential video content and in the fact that the storage nodes monitor their neighbors storage node to pinpoint failure.

These systems make use of RAID-style redundancy concepts for high reliability and availability, but concentrate on cluster environments as opposed to the grid/distributed environments explored in this work. In a cluster environment where all the nodes are owned by the same entity there is no need to implement a storage market.

### B. Grid Storage and Computation

Traditional cooperative grids, where resources are pooled within or among organizations, need to schedule resources in ways that are both efficient and redundant.

[4] presents a system for allocating fragments of databases, to provide high-bandwidth, high-parallelism and highly available access and processing. The sites are grouped by communication speed, and fragments are allocated in a way that reduces the computational cost of the potential access to the data. Redundancy of the fragments provides the increased availability. This

system, however, is not intended for resources that need to be purchased from providers, and does not attempt to minimize replication through monitoring of availability.

The Athena system [5] explicitly accounts for reliability in both node and links in a computational grid to tune the performance of a grid application. Athena performs a series of graph reduction algorithms to simplify the search for efficient transmission and execution paths. When a node wishes to run a program on a specific data set, Athena will use these graph reductions to calculate the most efficient path to transfer the data and the optimal nodes on which to execute the program, accounting for the reliability of the nodes and links. At runtime, it can then select the optimal path that is most likely to result in an efficient execution. While Athena was shown to be effective for execution in this environment, it does not explicitly account for pricing. It is not meant to allocate the data blocks, but merely to use the already allocated data blocks most efficiently.

### *C. Volunteer Computing and Storage*

The pooling of spare resources across the Internet, was started by highly specific computational projects such as distributed.net [6] and SETI@Home [7]. Over time, more generalized frameworks were developed, such as BOINC [8] and Bayanihan [9]. These systems are based on “volunteer computing,” using spare computation and storage resources to run important grand challenge type problems, typically from scientific disciplines.

However, the problem with volunteer computing, is that a user may submit false or corrupt data; a user may be highly motivated to make it look like he has done a lot more than he actually has. In [9], the author of the Bayanihan project discusses the idea of sabotage-resistant computing, proposing two ideas for combating this. The first is voting with N-modular redundancy. It is obvious that re-doing a computation becomes very expensive. The second idea includes spot checking with blacklisting. That is, check the solutions randomly, and ignore the results of those who do not pass the checks.

Freenet’s [10] primary purpose is ensure a confederated storage network that is resistant to being shutdown. Spare hard drive space and spare network bandwidth is utilized to store de-centralized web content. The content that is uploaded to this overlay network is replicated or replaced on the nodes in proportion to the content’s popularity. GUNet is a similar distributed storage project, intended to back up an individual user’s files onto the network [11].

The success of these projects has shown that both organizations as well as individuals possess spare computational and storage resources, and the contribution of such resources to large projects is technically feasible. However, when the altruistic motivation does not exist, it needs to be replaced with an interest based motivation, that is, with an economics of grid resources.

### *D. Grid Economics*

To achieve an efficient usage of spare computing resources, an efficient grid resource market, with sufficient liquidity is needed. Just like the stock of a company can be bought or sold nearly instantaneously, the participants in the computational resource market should be able to buy the desired resource without the need for complex search or negotiation.

Buyya, Abramson and Venugopal [12] discuss the shift from a system centric view of grid resource allocation, where parameters like throughput and utilization are optimized, to a view of the grid that takes the value of resources into account. They review the different economic models that have been explored in the context of grid resource allocation, including

commodity market models, posted price models, bargaining, contract-nets, auctions, cooperative bartering, monopoly and oligopoly.

The models and outlets for the future grid economy are still unestablished, and it is not clear what the community will adopt. However, much of the research into economic grid resource allocation makes reference to some sort of broker ([12]–[21] and others), suggesting that, no matter what the future grid economy looks like, brokers will play a significant role. At least in the infancy of applications that consume grid resources, allocating its own resources will place unnecessary strain and complexity on the system, which could be much more easily handled by a dedicated broker.

Because of the clear need of a broker, much research has centered around developing a broker architectures and markets. Gridbank [16], implements GASA (Grid Accounting and Services Architecture), which provides a variety of accounting and payment functions for resource brokering on the grid. It uses a service-oriented architecture to support grid applications, and keeps a database of producer/consumer accounts, and resource usage records. These usage records can help resource providers estimate prices for their resources. Gridbank supports three different types of payment methods including pay before use, pay as you go, and pay after use.

In [18], and [17], the authors present two complementary systems for economic resource allocation on the grid. The CPM (Compute Power Market) is intended to apply to low-end systems, while GRACE (Grid Architecture for Computational Economy), is intended for high end grid computing. For both systems, there are three entities, a market, a resource consumer and a resource provider. The resource providers register with the market and download a Market Resource Agents. Similarly, the resource consumers download a Market Resource Broker, that interacts with the market. The market itself mediates, potentially charging a fee.

In [13] a simple directory service called Grid Market Directory is deployed to allow both application and human users to browse and query available services by type, price and many other criteria. Future applications, for instance, looking for an image rendering service would find the lowest cost service and employ that service in the application. In [19] the proposed system offers different pricing schemes for general resources (such as CPU-time and storage) and specialized resources (such as access to a scientific instrument.) The system splits providers into groups, each coordinated by a separate broker. The brokers then coordinate to determine pricing for the resources. The specialized resources are priced through a double-auction.

The SX (Storage eXchange) system in [15] acts as a storage broker, allowing storage to be a tradeable resource. SX uses a double auction market model for open market trading and also allows storage to be exchanged. The system brokers storage requests by taking into account capacity, upload/download rate, and time frame of the storage reservation. The authors cite many other criteria that should be taken into account, including security, high-availability, fault-tolerance, reputation, consistency and operating environment.

Even though these architectures can match up resource consumers to providers, there is still the problem that none of the providers may be reliable enough, or have the correct scale of resources needed. For instance, if a pool of desktop users are providing the resources, then it is difficult to find enough storage or CPU power for large applications. Ideally, the market could compose disparate resources (storage, CPU) into requested ones.

Because grid resources can be an unreliable and intermittent resource, there may be a need for the broker or market to

compose a complex resource out a collection of simpler grid resources.

Mariposa [14] aims to execute database queries across distributed servers, using an economic framework. Each client with a query specifies a budget for the query, which is then passed to a broker. The broker communicates with the various data servers, who can trade data and queries at will. When the bidding processes completes, queries are finally executed and passed back to the user.

Oceanstore [22] aims to build a data storage infrastructure on untrusted servers, using both redundancy and cryptography. It is built around a market concept whereby users would pay a monthly fee for persistent and reliable storage. This would be supported by storage providers, at various locations who would trade storage resources among themselves. The prototype implementation of Oceanstore employs both Reed-Solomon and Tornado algorithms for encoding redundancy.

While Mariposa and Oceanstore compose the necessary resources from professional providers, they are not intended for millions of desktop web users to contribute resources. Individual resources, e.g. desktop PCs, must be composed differently than professional service providers. There must then be explicit accounting for availability. The proposed brokering algorithms aim to address these shortcomings, to open up to a larger list of sellers.

### III. AN ALGEBRA OF STORAGE RESOURCE COMPOSITION

The idea behind this paper is that users requests for storage with arbitrarily high capacity and availability requirement can be satisfied through the appropriate composition of cheaper resource components which, on their own, do not meet these requirements. However, we need appropriate guarantees that a specific composition indeed meets the requirements. In addition, we need an appropriate method to find the cheapest resource composition which satisfies a certain set of requests. To achieve this, we need a way to formally describe and manipulate composed storage resources, that is, we need an *an algebra of storage resource composition*. Table I summarizes the notations used in the remainder of this section.

We consider a simple or composed storage resource to be characterized by its capacity and availability. We define availability as the probability of successful resource access. We assume that a broker has control over a set of resources  $R_i$ ,  $i = 1 \dots n$ , each with a known capacity  $C(R_i)$  and availability  $A(R_i)$ .

A user requests resources from the broker by specifying the requested capacity  $C(R_{req})$  and availability  $A(R_{req})$ . These values are determined by the user based on the requirements of his application. A user will naturally accept a resource which has higher capacity and/or availability. The broker can satisfy the resource request in four different ways. In the simplest case, the request is satisfied with a simple resource  $R_{alloc}$ , which has the property  $A(R_{alloc}) \geq A(R_{req})$  and  $C(R_{alloc}) \geq C(R_{req})$ . The other three approaches are based on combining  $d$  distinct storage resources through additive composition ( $AC(R_1 \dots R_k \dots R_d)$ ), redundant composition ( $RC(R_1 \dots R_k \dots R_d)$ ) or distributed error correction ( $DEC(R_1 \dots R_k \dots R_d)$ ) to produce a composed resource  $R_{comp}$ .

Our approach took inspiration from Reliability Block Diagrams, adapting them to the concept of availability of distributed storage resources. The underlying assumption is that failures [23] in individual storage components do not stop the operation of other components.

*Additive composition* combines two or more smaller resources into one larger one. All the involved resources are required

TABLE I  
NOTATIONS FOR ANALYSIS

Notation	Description
$R$	Set of storage resources, each labeled $R_i$ where $1 \leq i \leq n$ . Each $R_i$ storage resource includes a capacity, an availability and a price. There is only one resource per network entity even though it may be divisible.
$C(R_i)$	Gives the storage capacity of the resource $R$ .
$A(R_i)$	Gives the availability probability of resource $R$ , $0 \leq A(R) \leq 1$
$P(R_i)$ :	Price per unit of storage for a storage resource.
$R_{req}$	A description of the resource required by the user.
$R_{comp}$	The resource being composed by the broker for the requesting user.
$d$	Number of resources chosen to be part of $R_{comp}$ .
$n$	Number of resources accounted for in the broker.
$AC(R_1...R_k...R_d)$	The result of additive composition of resources $R_1$ through $R_i$ .
$RC(R_1...R_k...R_d)$	The result of redundant composition of resources $R_1$ through $R_i$ .
$DEC(R_1...R_k...R_d)$	The result of distributed error composition of resources $R_1$ through $R_i$ .

to be available for the full storage resource to be considered available. The capacity and availability of an additively composed resource  $R_{comp} = AC(R_1...R_k...R_d)$  is given by the following formulas:

$$C(R_{comp}) = \sum_{k=1}^d C(R_k)$$

$$A(R_{comp}) = \prod_{k=1}^d A(R_k)$$

To additively compose a resource, the sum of the storage in the resources must be greater than or equal to our desired storage, and the product of the availabilities must be greater than or equal to the required availability.

As an example, suppose there are two available storage resources,  $R_1$  with 0.7GB and an availability of 0.95 and  $R_2$  with 0.3GB storage and 0.98 availability.

Therefore if both of these resources are used,  $C(R_{comp}) = 0.7GB + 0.3GB = 1.0GB$  and  $A(R_{comp}) = 0.98(0.95) = 0.931$ .

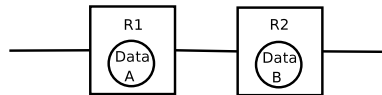


Fig. 1. Diagram of an additively composed resource.

*Redundant composition* is used whenever the user requires higher availability than offered by the currently available storage resources. For this kind of redundancy (known as N-modular redundancy) only one of the resources is required to work for a user to be able to access his data. When composing a redundant storage resource,  $R_{comp} = RC(R_1...R_k...R_d)$ , the capacity is constrained and the availability is given by

$$C(R_{comp}) = \min_{k=1...d} (C(R_k))$$

$$A(R_{comp}) = 1 - \prod_{k=1}^d (1 - A(R_k))$$

As an example, suppose there are two available storage resources,  $R_1$  with 1.0GB and an availability of 0.8 and  $R_2$  with 1.5GB storage and 0.85 availability. Therefore if both of these resources are used,  $C(R_{comp}) = \min(1.0GB, 1.5GB) = 1.0GB$

and  $A(R_{comp}) = 1 - ((1 - 0.8)(1 - 0.85)) = 0.97$ .

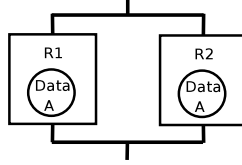


Fig. 2. Diagram of a redundantly composed resource.

The *Distributed Error Correction (DEC)* is analogous to the RAID5 disk drive composition method. This model must be composed of at least three resources. All but the last resource stores some portion of the data. The last resource stores the XOR of all of the other stores. In this way, the system can lose any one resource and still provide the data to the user. For availability all or all but one of the resources must be available. Therefore, composing a DEC resource,  $R_{comp} = DEC(R_1 \dots R_k \dots R_d)$ , with  $d$  individual resources, requires that

$$C(R_{comp}) = (d - 1) \min_{k=1 \dots d} (C(R_k))$$

$$A(R_{comp}) = \prod_{k=1}^d A(R_k) + \sum_{k=1}^d \left( (1 - A(R_k)) \prod_{j=1, j \neq k}^d A(R_j) \right)$$

As an example, suppose there are three available storage resources,  $R_1$  with  $C(R_1) = 0.7GB$  and  $A(R_1) = 0.8$ ,  $R_2$  with  $C(R_2) = 0.6GB$  and  $A(R_2) = 0.85$  and  $R_3$  with  $C(R_3) = 0.9GB$  and  $A(R_3) = 0.78$ .

If all three of these resources are combined into a DEC resource,  $C(R_{comp}) = 2 \min(0.7GB, 0.6GB, 0.9GB) = 1.2GB$ . For availability,  $A(R_{comp}) = 0.8(0.85)(0.78) + (1 - 0.8)(0.85)(0.78) + (1 - 0.85)(0.8)(0.78) + (1 - 0.78)(0.8)(0.85) = 0.9062$ .

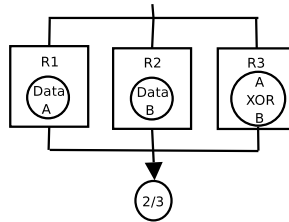


Fig. 3. Diagram of the DEC resource. In this instance, three storage resources are represented, with a reliability block diagram symbol representing the requirement that 2 out of 3 be operational.

It was previously stated that the user determines the requested resource capacity and availability from the requirements of his application. However, every application runs better with high capacity, highly available resources. To prevent applications requesting large amounts of resources with minimal benefits, the distributed storage system needs to implement an *economic model*. While the details of the economic models vary, all of them establish incentives for the clients to request resources close to the actual needs of the applications and, for the broker, incentives to satisfy the requirements as inexpensively as possible.

The brokers task, to satisfy as many requests as possible for the lowest possible price is the central challenge of the system.

#### IV. RESOURCE ALLOCATION ALGORITHMS

Our system assumes that a centralized broker mediates between the buyers and the sellers of the resources. As parties with resources to sell first come online, they register themselves with the broker according to a unique, persistent ID. The seller ID is used by the broker to keep track of whether the seller is online, the asking price of the resource, the parameters of the resource (size and availability), as well as its history of uptime, and transaction history. A buyer, also with a unique account ID, approaches the broker, requesting for a resource with specific capacity and availability.

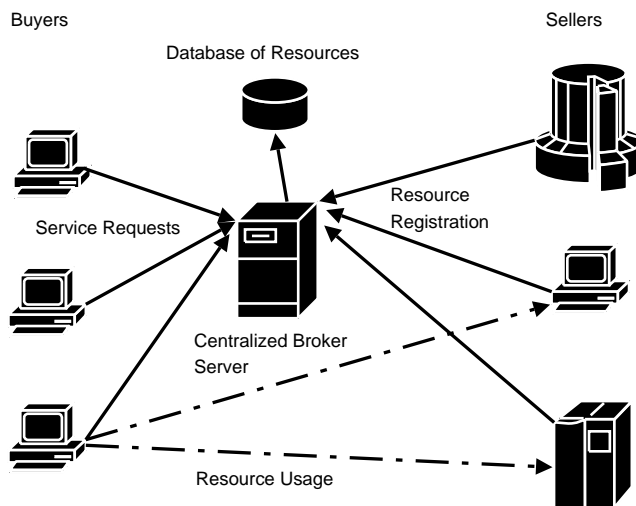


Fig. 4. Design of the system. The sellers may be home PC users, organizations with a large storage surplus or dedicated storage providers. Solid lines represent communication related to the brokering process, between buyers, sellers and the broker. The interrupted lines represent the peer to peer usage of the storage resources.

In our notation, the broker receives a number of requests, each labeled  $R_{req}^j$ . As with the resources mentioned in section III, each request has a value  $A(R_{req}^j)$  which gives the desired availability of the request and  $C(R_{req}^j)$  which gives the desired capacity of the request. The sellers which are registered with the broker are in a set  $R$  with the members labeled  $R_i$  ( $R_i \in R$ ). Each resource  $R_i$ , has an associated availability  $A(R_i)$ , capacity  $C(R_i)$ , and price  $P(R_i)$  per unit of storage. It should be noted that the needed storage portion can be separated from each  $R_i$ , leaving the rest for other allocations. The broker can meet requests by allocating portions of one or more of the resources in  $R$ . The broker may return an allocation that refers to portions of the storage resources from several different sellers, along with the the composition type of the allocation (single, AC, RC, or DEC). The resource  $R_{comp}$  refers to several individual resource allocations. Individual resource allocations include a seller and an amount of the resource to be used, given by the notation  $I_i = (R_i, C_{alloc}^i)$  where  $R_i$  is the seller's resource referred to and  $C_{alloc}^i$  is the amount of that resource allocated. As with the single resources, composed resources have associated availability  $A(R_{comp})$ , capacity  $C(R_{comp})$ , and price  $P(R_{comp})$ .

In the following we describe the algorithm used by the broker to compose and allocate resources. This brokering algorithm is composed of four other algorithms, designed to find a particular kind of allocation (standard, AC, RC or DEC). The main brokering algorithm applies these four algorithms to the current request and the current collection of sellers and chooses the least expensive resource that meets the requirements of the request (assuming a solution exists and can be found by the algorithms).

Let us start by introducing some notations used in the pseudocode:

- **SingleResource**, **RCResource**, **ACResource**, **DECResource**: the total resource allocations returned by the single resource search, the redundant composition resource search, the additive composition resource search, and the distributed error composition resource search, respectively. Each total resource allocation can consist of a number of individual resource allocations.
- **LowestCostResource**: the composed resource allocation that is found to be the lowest cost.

Next, we list the set of subroutines used by the broker's algorithm.

- **FINDLOWESTSINGLE**: Performs a linear search through the list of sellers to find the seller with the lowest price that still meets the minimum requirements.
- **FINDLOWESTRC**: Transforms the problem of finding the least expensive RC composition into the 0/1 knapsack problem, and solves it using a well known dynamic programming approach.
- **FINDLOWESTAC**: Uses a heuristic to find the lowest cost resource made by AC.
- **FINDLOWESTDEC**: Uses a genetic algorithm to find the lowest cost resource made by DEC.
- **SORTBYDIFFICULTY**: Sorts the list of currently queued requests by the difficulty criterion, that is  $C(R_{req}^j)/(1 - A(R_{req}^j))$  with easiest requests first.

```

SortByDifficulty( $R_{req}$ );
foreach  $R_{req}^j$  do
     $SingleResource = \text{FindLowestSingle}(R, R_{req}^j)$ ;
     $ACResource = \text{FindLowestAC}(R, R_{req}^j)$ ;
     $RCResource = \text{FindLowestRC}(R, R_{req}^j)$ ;
     $DECResource = \text{FindLowestDEC}(R, R_{req}^j)$ ;
     $LowestCostResource = \text{argmin } P(SingleResource), P(ACResource), P(RCResource), P(DECResource)$ ;
end

```

Fig. 5. Pseudocode for the broker's general algorithm.

The pseudocode for the general broker's algorithm is described in Figure 5. To begin with, the algorithm sorts all of its current requests, by the difficulty criterion  $C(R_{req}^j)/(1 - A(R_{req}^j))$ . This criterion is directly proportional to the requested capacity (larger requests have a larger difficulty criterion) and inversely proportional to  $(1 - A(R_{req}^j))$  (higher availability requests will also have a larger difficulty criterion). Dealing with difficult requests first ensures that they have the first pick of what is available, making it more likely that they can be met. Easier requests can then be met (even if at higher expense). Therefore, the broker can potentially meet more of its requests.

The general brokering algorithm searches through the available resources, as well as the possible composed resources, for resources that optimally meet the user's requirements. It applies four separate algorithms which are each designed to find allocations of different types. The algorithm then finds the least expensive of all the potential allocations (argmin in the pseudocode). We now present each of the search functions in turn.

### A. Redundant Composition Algorithm

The first algorithm is designed to find whether there are any RC compositions possible to meet the current request among the available resources and if so, which one is the least expensive. First, we show that the problem of finding the optimally priced redundantly composed storage resources can be reduced to the classical knapsack problem. The problem is cast as follows. There are several items  $i$ , numbered from 1 to  $n$  to be placed in a knapsack of capacity  $c$ . Each item has an associated price,  $p_i$  and an associated weight,  $w_i$ . The goal is to maximize the value of the knapsack contents while staying within the capacity. If we assume that an item must be taken or not taken, the problem is then referred to as the 0-1 Knapsack problem.

Redundant composition can be reduced to the knapsack problem. The items map to the storage resources,  $R_i$ , and the price  $p_i$  is instead  $P(R_i)$ . The capacity  $c$  and the weights  $w_i$  are represented by expressions of the availability.

First, all sellers that have  $C(R_i) \geq C(R_{req})$  are selected. Because the full data must be replicated when it is stored with RC, it is clear that any resource without the full requested capacity  $C(R_{req})$  can be automatically excluded. Using the equation of availability for redundant composition:

$$1 - \prod_{k=1}^d (1 - A(R_k)) \geq A(R_{req})$$

easily becomes

$$\prod_{k=1}^d (1 - A(R_k)) \leq (1 - A(R_{req}))$$

For reasons that will become clear later, both sides must be greater than 1. A good way to guarantee this is by multiplying by

$$\frac{1}{(1 - \max(A(R_k)))^d}$$

This yields:

$$\prod_{k=1}^d \frac{1 - A(R_k)}{(1 - A(R_{max}))^d} \leq \frac{1 - A(R_{req})}{(1 - A(R_{max}))^d}$$

Now, this equation is turned into a linear weight function, by taking the natural logarithm.

$$\sum_{k=1}^d \ln \left( \frac{1 - A(R_k)}{(1 - A(R_{max}))^d} \right) \leq \ln \left( \frac{1 - A(R_{req})}{(1 - A(R_{max}))^d} \right)$$

If the term  $x_i$  is added either equal to 0 or 1, depending on whether the resource is included, and added over all  $n$  instead of  $d$ , the problem is now the 0/1 knapsack problem, with a weight function  $w_i = \ln \left( \frac{1 - A(R_k)}{(1 - A(R_{max}))^d} \right)$  and  $c = \ln \left( \frac{1 - A(R_{req})}{(1 - A(R_{max}))^d} \right)$ .

$$\sum_{i=1}^n x_i \ln \left( \frac{1 - A(R_i)}{(1 - A(R_{max}))^d} \right) \leq \ln \left( \frac{1 - A(R_{req})}{(1 - A(R_{max}))^d} \right)$$

While the aim for the original knapsack problem is to maximize the price of the included items, our aim is to minimize price. We can easily adapt the knapsack algorithm used to minimize price instead.

The general knapsack problem is NP-hard and the decision version is NP-complete [24]. However, for integer knapsack capacity, a pseudo-polynomial approach does exist that uses dynamic programming [25]–[27]. To find an optimal packing takes time  $O(nc)$  [24]. If the weight functions  $w_i$  and  $c$  are mapped from their real values into integer weights, then the dynamic-programming-based knapsack algorithm can be used to approximately find the optimal RC allocation. However, for the equations to become correct, we must try sufficient numbers of  $d$  from 2 to  $n$  in the expressions  $w_i = \ln\left(\frac{1-A(R_k)}{(1-A(R_{max}))^d}\right)$  and  $c = \ln\left(\frac{1-A(R_{req})}{(1-A(R_{max}))^d}\right)$ . Therefore, the final RC allocation algorithm iterates over values of  $d$  from 2 to  $n$  and then applies the dynamic-programming-based knapsack algorithm to the seller resources. This algorithm allows the problem to be solvable in pseudo-polynomial time (dependent not only on the number of resources  $n$ , but on the value  $c$ )  $O\left(n^2 \log \frac{1-A(R_{req})}{(1-A(R_{max}))^n}\right)$ .

The dynamic programming algorithm for solving the knapsack problem works as follows. Begin with a table of weights  $W[1..n, 0..c]$  where  $n$  is the total number of sellers who are eligible to be included (have enough capacity) and  $c$  is the integer representation of the availability quantity  $\ln\left(\frac{1-A(R_{req})}{(1-A(R_{max}))^d}\right)$ . Every value  $W[i, j]$  in the table will contain the maximum value that can be included if  $c = j$ . Looking in this table will reveal the appropriate resources to be included. For more information on this classical solution to the knapsack problem see [24]–[27].

### B. Additive Composition Algorithm

To find the optimal additively composed resource, a second constraint is added to the knapsack problem. Suppose that, in addition to not exceeding the capacity of the sack, one must not exceed another arbitrary dimension either (e.g. length). This is referred to as the 2 dimensional knapsack problem, which can also be extended to an arbitrary dimensionality, becoming the d-dimensional knapsack problem.

For additive composition, the resources that are selected when put together, must meet or exceed  $C(R_{req})$ . The composed resource also must not exceed the availability constraints. As stated before, the availability that must be met is

$$\prod_{k=1}^d A(R_k) \geq A(R_{req})$$

By taking the reciprocal and the natural logarithm, we yield an arrangement that is compatible with the knapsack weight equation, as before:

$$\sum_{k=1}^d \ln\left(\frac{1}{A(R_k)}\right) \leq \ln\left(\frac{1}{A(R_{req})}\right)$$

However, we must add the additional constraint that

$$\sum_{i=1}^d C(R_i) \geq C(R_{req})$$

This yields a 2-dimensional knapsack problem, which is not easily solvable. Therefore, we design a heuristic that takes into account knowledge of the domain.

To begin developing a heuristic, we first cull out resources that cannot be part of this kind of allocation. It is known that all sellers with  $A(R_i) < A(R_{req})$  can be automatically excluded, because they would immediately cause the availability of the allocation to drop below  $A(R_{req})$ . The AC algorithm then begins iterating through the remaining resources, starting with ones that are heuristically determined to be more promising. We now devise a metric that gives preference to resources that would be better suited for AC allocations, called the AC criterion.

The AC criterion is defined to be:

$$\sigma(R_i, R_{req}^j) = \frac{A(R_i)}{P(R_i)} \cdot \left( \frac{\min(C(R_i), C(R_{req}^j))}{C(R_{req})} \right)$$

The first part of the product  $\frac{A(R_i)}{P(R_i)}$  gives the availability per unit price. This term is included give more emphasis to resources that have good availability relative to their cost. Having a somewhat higher availability is important in AC allocations, because availability of the composed resource drops as the product of its constituents ( $\prod_{k=1}^d A(R_k) \geq A(R_{req})$ ). The second part of the term  $\left( \frac{\min(C(R_i), C(R_{req}^j))}{C(R_{req})} \right)$  becomes a factor from  $[0, 1]$  giving more emphasis to sellers with large allocations available. However, the *min* function limits this influence to  $C(R_{req})$ , because there is no advantage to having more space than is required when finding an AC allocation. This term helps find fewer large resources, instead of many smaller resources to again avoid a rapid reduction in availability.

- **FINDALLGREATEROREQUAL**: returns a list of sellers with availability greater than or equal to the specified availability.
- **SORTBYSTRIPINGCRITERION**: sorts the list descending by the calculated  $\sigma$ .
- *currentAllocation*: A composed resource made of a set of individual resource allocations.

```

Procedure: FindLowestAC
( $R, R_{req}^j$ )
sellersWithMinA = FindAllGreaterOrEqual( $R, A(R_{req}^j)$ );
SortByStripingCriterion(sellersWithMinA,  $C(R_{req}^j)$ );
real accumulatedSpace = 0;
real accumulatedAvailability = 1;
boolean done = false;
boolean valid = true;
currentAllocation =  $\emptyset$ ;
while  $\exists R_i \in$  sellersWithMinA and done  $\neq$  true do
  if accumulatedSpace +  $C(R_i) \geq C(R_{req}^j)$  then
    done = true;
  end
  if accumulatedAvailability *  $A(R_i) < A(R_{req}^j)$  then
    done = true;
    valid = false;
  end
  if valid then
    currentAllocation = currentAllocation  $\cup$   $R_i$ ;
    accumulatedSpace +=  $C(R_i)$ ;
    accumulatedAvailability *=  $A(R_i)$ ;
  end
end
return currentAllocation;

```

Fig. 6. Heuristic algorithm for finding an additively composed set of resources.

The algorithm takes a set of seller resources  $R$  and the requested resource  $R_{req}^j$ . As stated before, sellers without the minimum availability are first culled out of the group because they would immediately bring the availability of allocation below the requested level. Then the algorithm sorts the remaining resources in descending order by the defined AC criteria. Following the sort, the algorithm iterates through and includes available resources until it has found a satisfactory composed resource, or until there are no more resources with the minimum availability remaining. For each resource, the algorithm checks to see if its addition would help the resource meet the necessary requirements. If a valid resource is found, the algorithm then stops searching and returns the resource. Otherwise, it will return a null resource.

The loop through each of the  $R_i$  is an  $O(n)$  process, while the sort is an  $O(n \log n)$  process, making this algorithm work in  $O(n \log n)$ .

### C. Distributed Error Composition Algorithm

Finding the optimal distributed error composition configuration is a complex integer programming problem. However, there are a number of meta-heuristics to which we can turn. Tabu search and genetic algorithms have both been successfully applied to knapsack problems [28]. While our problem is harder than the traditional knapsack problem, it is relatively easy to frame in terms of a genetic algorithm. The chromosome is formed with a binary gene for each available seller. The GA then tries to maximize a fitness function which has been engineered to reward solutions meeting the minimum size, whose sellers have the minimum required capacity and solutions that meet the requested availability. The fitness function is defined as follows:

$$Fitness(DEC) = \frac{D_1}{A(R_{req})} \min(A(DEC), A(R_{req})) + D_2 \left(1 - \frac{d_{suff}}{d}\right) + D_3(ProperNumber(d)) + \frac{D_4}{P}$$

where  $D_1$ ,  $D_2$ ,  $D_3$  and  $D_4$  are positive constants,  $P$  is the price per unit of the entire allocation,  $d$  is the number of total sellers in the allocation and  $d_{suff}$  is the number of resources with sufficient space to be part of the allocation. *ProperNumber* rewards solutions that are within a probable range of valid DEC solutions and is defined as follows:

$$ProperNumber(d) = \begin{cases} 1 & 3 \leq d \leq DEC\_MAX \\ 0 & otherwise \end{cases}$$

*DEC\_MAX* (in our case 10) is a parameter of the broker representing the practical limit of the number of different  $R_i$  resources that can be part of  $R_{req}$ . The the number of storage resources  $d$  that are part of  $R_{req}$  could conceivably be as large as  $n$ , but *DEC\_MAX* is used to find practical solutions and computationally simplify the task. Because this GA can produce individuals that are not valid solutions, they are checked for validity before being added to the list of possible solutions.

The fitness function of the genetic algorithm used for DEC must find solutions that are not only valid (fulfill the requested availability  $A(R_{req})$ ) but minimize the price. Early experiments which only selected on the criteria of price had trouble finding valid solutions. Therefore, additional terms are added to the fitness function which reward solutions for being closer to correct. In addition to the fourth term, which selects for price, terms one to three help bias the search toward valid solutions.

The first term of the fitness function,  $\frac{D_1}{A(R_{req})} \min(A(DEC), A(R_{req}))$ , evaluates how close the solution comes to meeting the requested availability  $A(R_{req})$ . By taking the minimum of  $A(DEC)$  and  $A(R_{req})$ , it ensures that the value remains between

$[0, 1]$ . From the standpoint of the broker, there is no need to obtain a solution with greater than requested availability. The constant  $D_1$  provides a bias for this criterion.

The second term is designed to find solutions where all of the included seller resources have enough capacity to support the requested resource. Allocations where the included resources do not all have enough capacity are technically invalid, but rewarding closer solutions helps the GA concentrate its search. Dividing the number of sufficient-capacity resources over the total included resources, determines what percentage of the resources have sufficient capacity.

The third term is designed to ensure that the solution contains a reasonable number of different sellers. Because redundancy in a DEC allocation is entirely contained within one extra allocation, availability will quickly drop for allocations of too many sellers. The term ensures that the number of sellers in the allocation meets the minimum number of sellers for DEC (3) and is less than or equal to the maximum number of sellers ( $DEC\_MAX$ ). While solutions that are not within this range are technically invalid, giving them no fitness discourages the genetic algorithm from pursuing solutions that are close to correct.

The fourth and final term helps the genetic algorithm search for smaller prices. Finding the lowest price (assuming that a valid solution has been found) is the main objective. Therefore, smaller prices will drive the fourth term to be larger, with appropriate bias by  $D_4$ .

## V. EXPERIMENTAL STUDY

### A. Experimental setup

In the following we describe an experimental study which measures the benefits of our proposed approach compared to a standard approach which involves only simple resources. We assume that the broker queues up a batch of requests and clears them in a single allocation step. This way more efficient allocations can be made compared to the case when a single requests is allocated at a time. In our experiments we assumed that a batch of 50 requests are cleared in an environment with 200 sellers.

We compared two approaches:

- Standard allocation: the broker performs a linear search on the single resources to meet the requests.
- Improved allocation: the broker performs a search on the single resources and the improved search algorithms described in Section IV for the three types of composed resources.

The data sets considered in our experiments were taken from the work of Anderson and Fedak [29] concerning the statistical properties of the hosts participating in the SETI@Home project. We generated the list of storage resources available was generated by randomly selecting 50,000 free disk space  $d\_free$  values from the host database. We filtered out values above  $10^{14}$  bytes, as these are probably due to erroneous reporting. Unfortunately, there is no host availability data provided on per-host basis by the publicly available host database. Therefore, we generated some artificial values which match the statistical properties of the SETI@Home hosts. We considered three reported values:  $on\_fraction$ , the fraction of time that the SETI@Home client runs, the  $connected\_fraction$ , the amount of time that the client program has access to an Internet connection and finally, the  $active\_fraction$  shows the amount of time that the client is allowed to run. For our application, we consider the average availability to be the product of these three variables:

$$\begin{aligned}\bar{A} &= (on\_fraction)(connected\_fraction)(active\_fraction) \\ &= (0.81)(0.83)(0.84) = 0.5647\end{aligned}$$

Using the value as the mean of a Gaussian random distribution with a standard deviation of 0.1, availability values were generated and paired with randomly selected disk space values. Finally, prices per Gigabyte of space were generated for each resource pair  $R_i$ . We assume that the pricing per unit of storage space is dictated by the availability, which becomes asymptotically more expensive as availability approaches one:

$$P \sim \frac{1}{1 - A(R_i)} \quad (1)$$

To provide random variations in pricing, this calculated price is multiplied by a normally generated random price factor with average of 1.0 and standard deviation of 0.1.

For each clearing iteration within a run with particular values for the two parameters, 200 sellers were randomly selected from the loaded seller pool, and 50 consumers were randomly generated. For each set of parameters, this clearing iteration was performed 10 times and averaged. The consumers were randomly generated with average requested availabilities  $\bar{A}(R_{req})$  in the range of 0.3 to 0.99 and average requested capacity  $\bar{C}(R_{req})$  of 1GB, 10GB, 100GB and 300GB. With these parameters, consumer requests were generated by Gaussian random generators, with the given averages and standard deviations of  $\frac{\bar{A}(R_{req})}{8}$  and  $\frac{\bar{C}(R_{req})}{2}$  for the required availability and capacity respectively. These were chosen under the assumption that the availability required by users would not vary as widely as the space required (most users want “pretty good” availability). The GA responsible for finding Distributed Error Compositions run for a 100 generations with a population size of 100. The default JGAP mutation rate of 0.1 was used. The constants for the fitness function were,  $D_1 = 25.0, D_2 = 25.0, D_3 = 200.0, D_4 = 50.0$ .

## B. Results

Figure 7 shows the percentage of allocation successes for four different average request sizes. We plot both the standard and the improved algorithm. We expect the improved algorithm to perform at least as well as the standard algorithm, as the improved algorithm subsumes the previous one. The question is whether the improvement in the allocation success justifies the considerable computational expense of the improved algorithm.

The first observation, applicable to both the standard and the improved algorithm, is that the success rate becomes lower with the increase on the average requested availability  $\bar{A}(R_{req})$ , as the requests become increasingly hard to meet. Second, the higher the average requested capacity  $\bar{C}(R_{req})$ , the lower the success rate.

For cases when the both average requested availability and requested capacity is low, both algorithms can guarantee close to 100% success rate. For other cases, however, the improved algorithm considerably outperforms the standard algorithm. In fact, for a considerable range of scenarios, the improved algorithm can maintain success rates close to 100% even when the success rate of the standard algorithm is very low. For instance, for requests with average capacity of 10GB and availability 0.9, the success rate of the improved algorithms is close to 100%, while for the standard algorithm is around 20%.

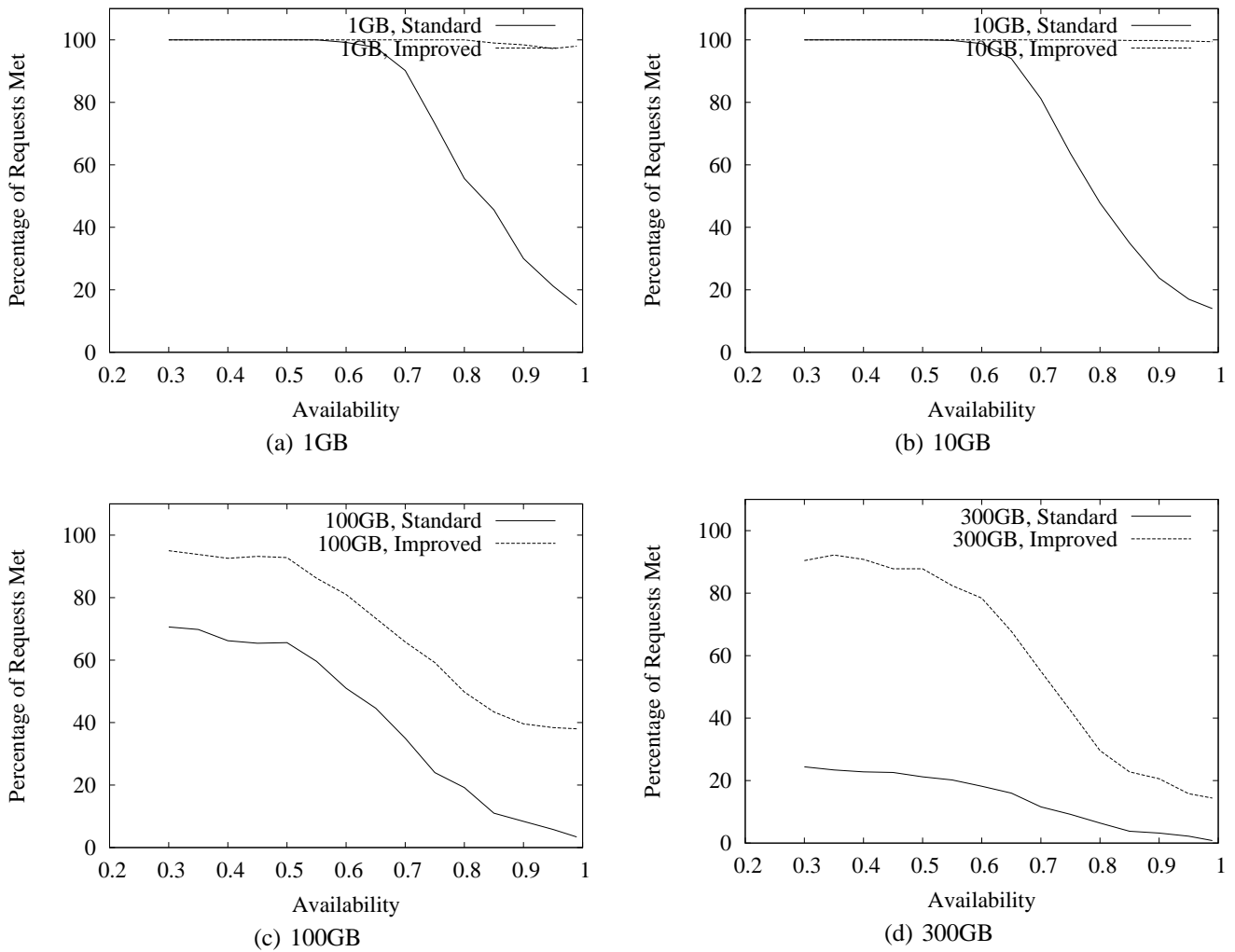


Fig. 7. Percentage of successful allocations, using the standard and the improved allocation algorithm, for various average request sizes.

Still, the success rate of the improved algorithm declines for scenarios which have both high requested capability (100GB and above) and high requested availability (0.7 and above). Even in these cases, the improved algorithm outperforms the standard algorithm with a significant margin, which justifies the additional computational resources.

Figure 8 presents the price of the allocated resources function of the requested average availability and for various values of the average requested size. Both algorithms try to minimize the cost of the allocated resources. As the improved algorithm subsumes the standard one, the average prices for the improved algorithms will be at least as low as that for the standard one.

The overall space of the graphs reflects, in broad lines, the pricing model of Formula 1. Nevertheless, the average price for satisfying the customer requests can be higher or lower than the one given by the price function under the following conditions:

- The price will be higher if there are not enough resources to satisfy the requests at the desired availability level and the broker needs to satisfy it with higher availability, more expensive resources.
- The price may be lower if the broker is able to satisfy the request using redundant or DEC composition. This is true only for certain types of size, availability and composition type combinations.

These considerations show that the improved algorithm should be able to guarantee lower prices. Indeed, the graphs in Figure

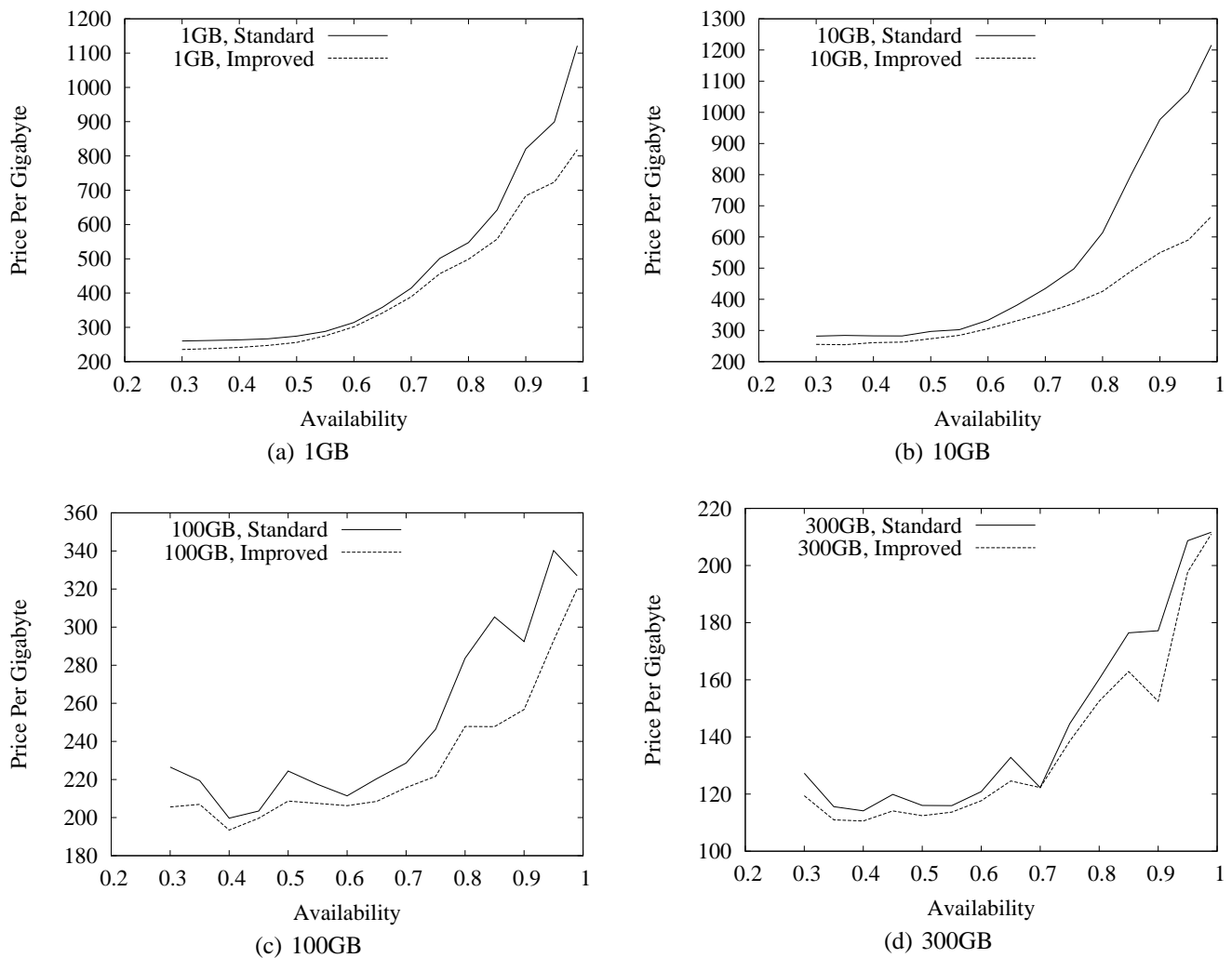


Fig. 8. Prices for allocations with single and multiple resources.

8 validate this conjecture. However, the difference between the methods is significant only in the cases when the requests have a high average availability and are of a medium 10-100GB size, where the savings can be as high as 50%.

Next, let us study the relative contribution of the different resource allocation types to the satisfaction of the requests. Our objective, is again, to study whether the improved algorithm is justified. If a very large percentage of requests are satisfied using single resource allocations, then the additional computational complexity of the improved method is not justified. The graphs in Figure 9, however, show that this is not the case. In particular for the “difficult” allocation cases (with high availability and large capacity requests) most of the requests are fulfilled with composed resources (mostly DEC and Redundant). For small average requested sizes, standard allocations at low availability can be cheaper, and both DEC and standard allocations require less redundancy (and therefore, potentially less cost) than redundant composition. This makes them dominant at low required availabilities. For the larger requested sizes (in Figure 9c and 9d), the standard allocations are not as dominant, even at low required availabilities. This is because as the average requested size becomes larger, large enough single resources are less likely to be found, and DEC has the most efficient redundancy. As availability requirements are more stringent, anything but redundant composition has less and less chance of meeting it.

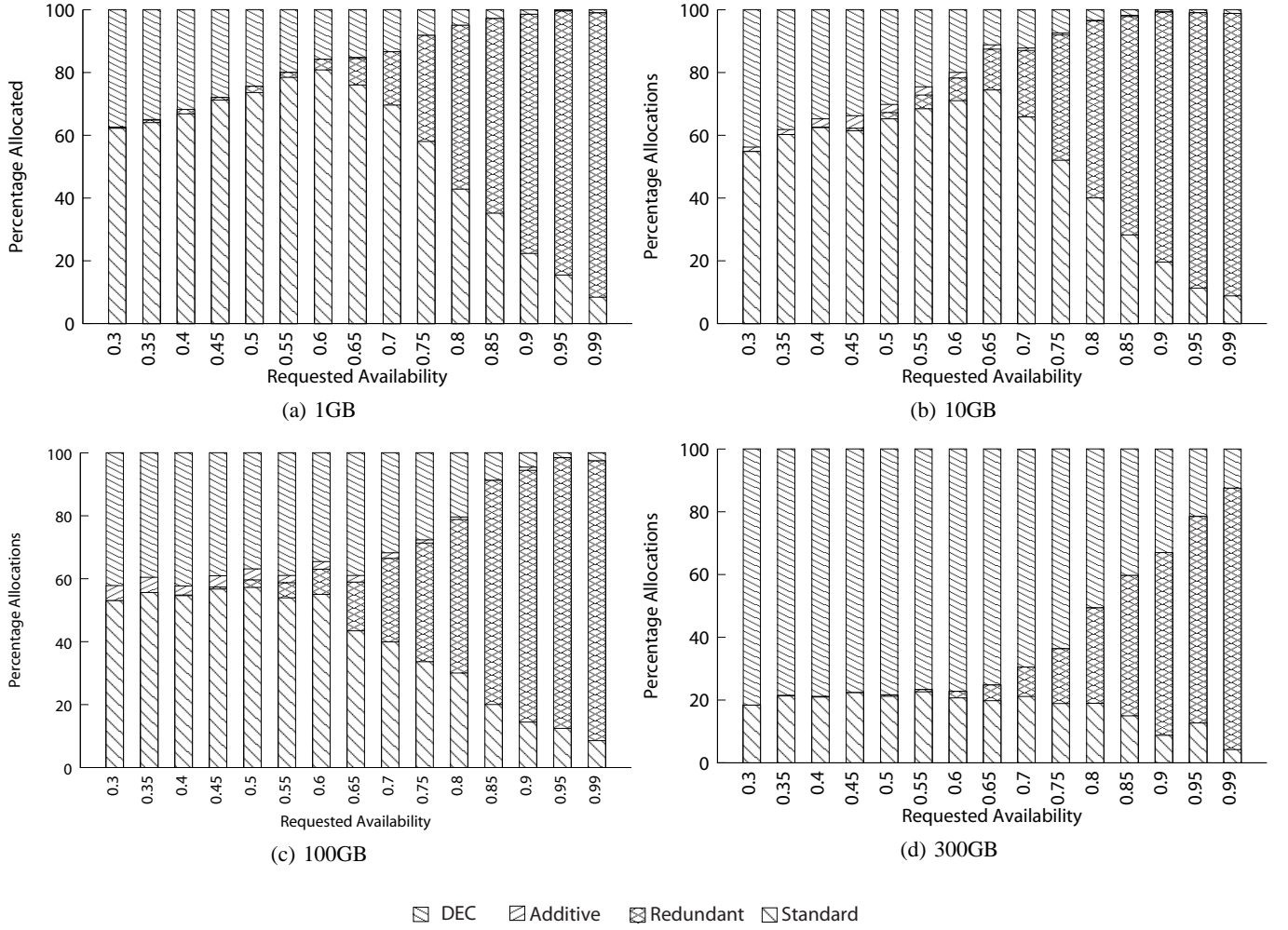


Fig. 9. Percentage of compositions for runs at different average request sizes.

We note that only a very small percentage of the requests were fulfilled using additive composition. This is due to the fact that the availability of an additively composed resource declines rapidly, and the average availability of the seller host pool is already somewhat low. It is also more rare, especially at small sizes, to find less expensive additive compositions because it takes several higher availability resource to make one lower availability resource.

Because the genetic algorithm used in finding the optimal DEC allocations was relatively successful, it is worth finding out how well the algorithm and the associated parameters would work in different grid environments. To evaluate this, we performed tests for a single request of  $A(R_{req}) = 0.65$  and  $C(R_{req}) = 300GB$  averaged over 50 iterations. The sets of sellers used were artificially generated to have average capacities of 25GB, 50GB, 75GB and 100GB. Each set of sellers was evaluated over the values from 1 to 500 of each parameter. While each parameter of the fitness function was swept over its domain of values, the others were kept constant at the levels used in our prior tests ( $D_1 = 25$ ,  $D_2 = 25$ ,  $D_3 = 200$ ,  $D_4 = 50$ ). The figures 10a, b, c and d illustrate the graphs for  $D_1$ ,  $D_2$ ,  $D_3$  and  $D_4$  respectively.

The graphs show that the optimal fitness values remain relatively constant despite market environments with differing collections of resources. This stability indicates that the values chosen will likely translate well to other grid environments. However, for any application of this genetic algorithm, it would likely behoove the developers to simulate system behavior

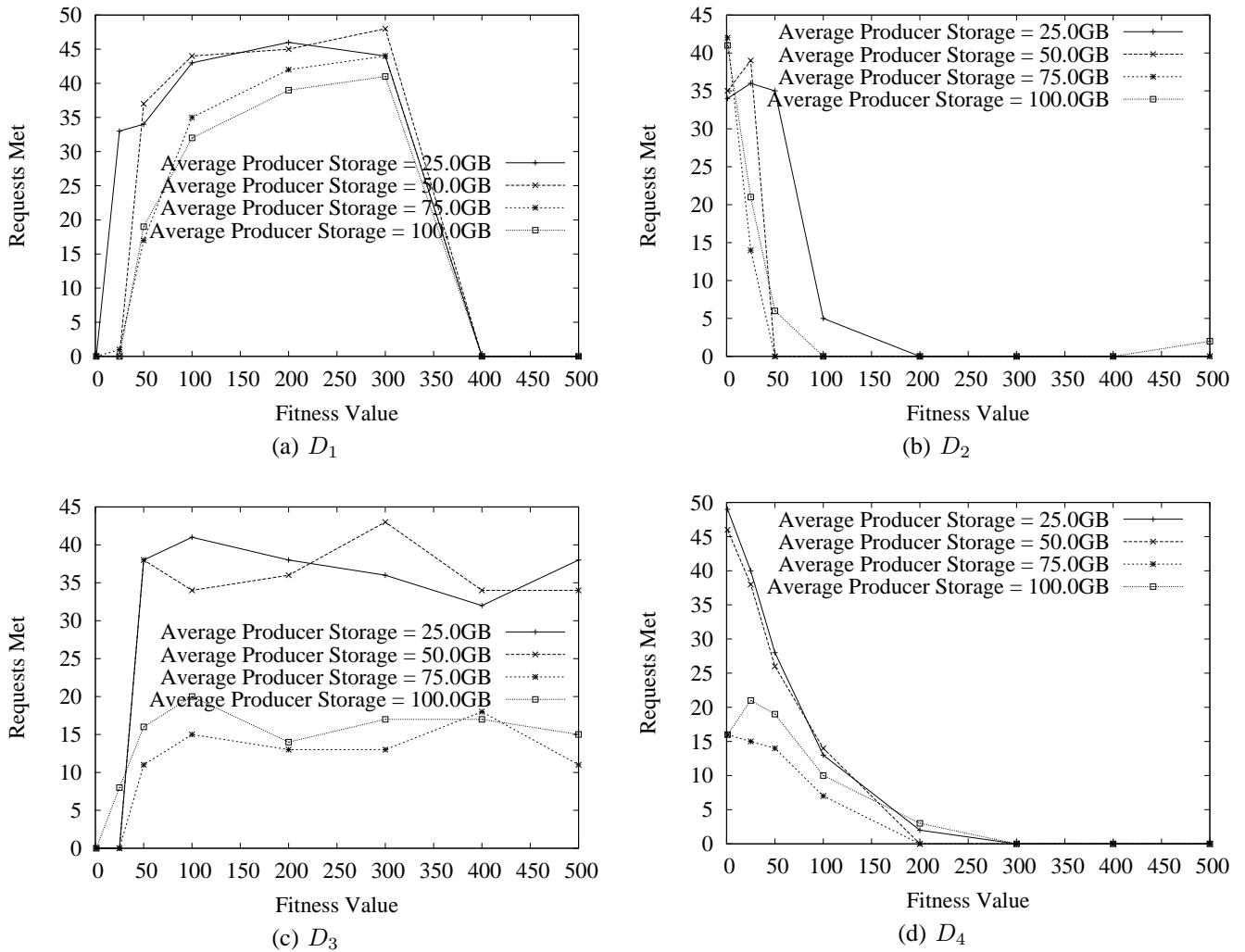


Fig. 10. Varying fitness function values for a request of  $A(R_{req}) = 0.65$  and  $C(R_{req}) = 300GB$

using known distributions of sellers and requests for the environment at hand. These simulations can ensure that the broker delivers maximum performance for the environment in which it is used.

In figure 11, we show the time required to execute the various algorithms. The experiments were executed on a single core of an 2.2GHz Intel Core2 Duo with 2GB of RAM. The simulations were in Java with no effort of optimization. The improved algorithm is broken down by its constituent allocation searches to better indicate which phases are the most intensive.

In figure 11, the timing for the standard algorithm and the additive composition algorithm are absent, because they took less than a millisecond to meet each request. Not surprisingly, the extra analysis came at the expense of additional computation. The redundant composition algorithm took an average of about 50ms for 100 sellers to about 550ms for 1000 sellers. The genetic algorithm for DEC took from about 1.2s per allocation for 100 sellers to over 10s per allocation for 1000 sellers. While this is significantly greater than the standard algorithm, it can still be practical. For more permanent allocations, it may be worth both the client's and the broker's investment of time to find an allocation that is less expensive. Additionally, the algorithm can potentially be parallelized to greatly accelerate it.

As a final evaluation of the new algorithm, we execute a known test problem, which is small enough to find the optimal

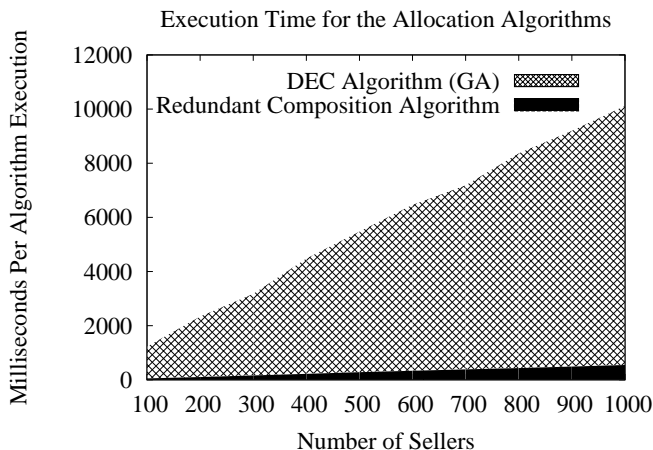


Fig. 11. Execution time of the different search methods in the improved algorithm. The times are not included for the Additive Composition and Standard algorithms because they were less than 1.0ms for all tests executed.

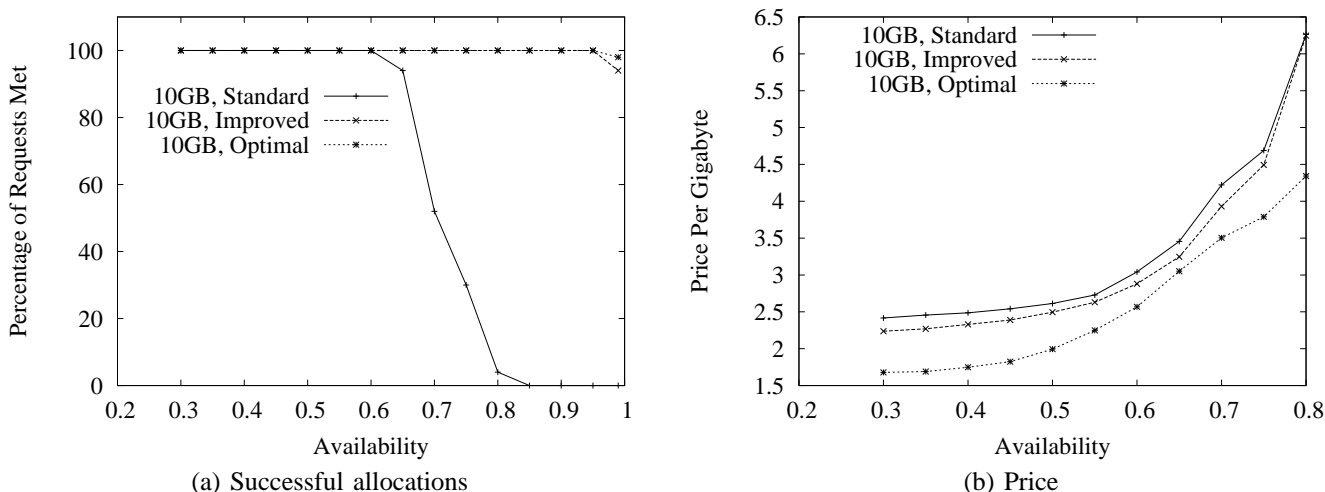


Fig. 12. Comparison of the standard, improved and optimal algorithms for  $C(R_{req}) = 10GB$  and 15 sellers.

answer. As has been stated before, because the allocation problems are in NP, finding the guaranteed optimal solutions requires a brute force search, taking a time exponential in the number of sellers. Therefore, any comparison to the optimal solution would have to be with a relatively small problem. To do such a comparison, a broker was evaluated with 15 sellers over a simple request. A request of 10GB with varying requested availability was made to the broker. This was done over 50 iterations (for 50 different sets of 15 sellers) and averaged. The results are shown in figure 12.

In 12a, the percentage of successful allocations is illustrated. Despite the standard algorithm rapidly losing the ability to make an allocation after a requested availability of about 0.65, the improved algorithm remains either at or just below optimal. For the pricing in 12b, pricing information was only compared as far as all three algorithms could find some solution, otherwise those prices were not counted. As expected, the improved algorithm's pricing remained between the standard algorithm and the optimal algorithm. This fact demonstrates that while there improved algorithm does offer savings over the standard algorithm, there is still room for improvement and future work.

## VI. CONCLUSIONS/FUTURE WORK

In this paper we proposed an approach to combine cheap network storage resources to achieve the desired availability and capacity requirements of consumers. Although the optimal allocation problem is NP-hard, the combination of several approximation techniques can be used by the storage brokers improve the service provided to the users. Allocation algorithms such as the ones presented may serve as the core of resource brokers on a future grid resource market.

Our future work involves both improving the performance of the proposed solutions, as well as extending them to a more general problem. An important extension is to consider the network bandwidth through which the storage is available. Co-allocation strategies like the one presented in [21] may reduce the transmission times of the involved blocks. The network bandwidth should be factored in to efficient allocation as in, [20], and should affect pricing.

With the use of better representations and better evolutionary algorithms, it may be possible to compose more complex arrangements of resources than the three presented. In addition, resources such as CPU power should be accounted for in applications requiring more than just data storage. Such additional criteria can fit relatively easily into a genetic algorithm framework.

## ACKNOWLEDGMENT

This work was supported in part by NSF grants: 0341601, 0647018, 0717674, 0717680, 0647120, 0525429, 0203446, as well as an NSF Graduate Research Fellowship. The authors would also like to acknowledge Advanced Micro Devices, Inc. (AMD) for their generous donation of equipment used in this research.

## REFERENCES

- [1] V. Lindenstruth, R. Panse, T. Steinbeck, H. Tilsner, and A. Wiebalck, "Remote administration and fault tolerance in distributed computer infrastructures," in *Future Generation Grids*, V. Getov, D. Laforenza, and A. Reinefeld, Eds. Springer, 2005, pp. 61–80.
- [2] A. Bonhomme and L. Prylli, "Reconfiguration of RAID-like data layouts in distributed storage systems," in *Proceedings of the 2004 International Parallel and Distributed Processing Symposium*, April 2004, pp. 212–219.
- [3] —, "Performance evaluation of a distributed video storage system," in *Proceedings of the 2005 International Parallel and Distributed Processing Symposium*, April 2005, pp. 126–135.
- [4] I. O. Hababeh, M. Ramachandran, and N. Bowring, "A high-performance computing method for data allocation in distributed database systems," *Journal of Supercomputing*, vol. 39, no. 1, pp. 3–18, January 2007.
- [5] H. Jin, X. Xie, Y. Li, Z. Han, Z. Dai, and P. Lu, "A real-time performance evaluation model for distributed software with reliability constraints," *Journal of Supercomputing*, vol. 34, no. 2, pp. 165–179, November 2005.
- [6] "Distributed.net," <http://distributed.net/>.
- [7] "Seti@home," <http://setiathome.berkeley.edu/>.
- [8] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, November 2004*.
- [9] L. F. G. Sarmenta, "Sabotage-tolerance mechanisms for volunteer computing systems," in *Proceedings of the ACM/IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01), May 2001*.
- [10] "The free network project," <http://freenetproject.org/>.
- [11] C. Grothoff, "An excess-based economic model for resource allocation in peer-to-peer networks," *Wirtschaftsinformatik*, vol. 45, no. 3, pp. 285–292, 2003.
- [12] R. Buyya, D. Abramson, and S. Venugopal, "The grid economy," *Proceedings of the IEEE, Special Issue on Grid Computing*, vol. 93, no. 3, pp. 698–714, 2003.

- [13] J. Yu, S. Venugopal, and R. Buyya, "A market-oriented grid directory service for publication and discovery of grid service providers and their services," *Journal of Supercomputing*, vol. 36, no. 1, pp. 17–31, January 2006.
- [14] M. Stonebraker, P. Aoki, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu, "Mariposa: A wide-area distributed database system," *Very Large Databases (VLDB)*, vol. 5, no. 1, pp. 48–63, January 1996.
- [15] M. Placek and R. Buyya, "Storage exchange: A global trading platform for storage services," in *Proceedings of the 12th International European Parallel Computing Conference (EuroPar 2006)*. Springer-Verlag, August 2006.
- [16] A. Barmouta and R. Buyya, "GridBank: A grid accounting services architecture (GASA) for distributed systems sharing and integration," in *Workshop on Internet Computing and E-Commerce, Proceedings of the 17th Annual International Parallel and Distributed Processing Symposium (IPDPS 2003)*, April 2003.
- [17] R. Buyya, D. Abramson, and J. Giddy, "An economy driven resource management architecture for global computational power grids," in *The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, June 2000.
- [18] R. Buyya and S. Vazhkudai, "Computer power market: Towards a market-oriented grid," in *The 1st IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2001)*, May 2001, pp. 574–581.
- [19] C. Weng, M. Li, and X. Lu, "Grid resource management based on economic mechanisms," *Journal of Supercomputing*, vol. 42, no. 2, pp. 181–199, November 2007.
- [20] C. Yang, P. Shih, C. Lin, and S. Chen, "A resource broker with an efficient network information model on grid environments," *Journal of Supercomputing*, vol. 40, no. 3, pp. 249–267, June 2007.
- [21] C. Yang, I. Yang, K. Li, and S. Wang, "Improvements on dynamic adjustment mechanism in co-allocation data grid environments," *Journal of Supercomputing*, vol. 40, no. 3, pp. 269–280, June 2007.
- [22] J. Kubiawicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000, pp. 190–201.
- [23] "System reliability theory & principles reference from ReliaSoft," <http://www.weibull.com/systemrelwebcontents.htm>.
- [24] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*. Springer, 2004.
- [25] R. Bellman, *Dynamic Programming*. Princeton University Press, 1957.
- [26] R. Garfinkel and G. Nemhauser, *Integer Programming*. John Wiley and Sons, 1972.
- [27] T. Hu, *Integer Programming and Network Flows*. Addison-Wesley, 1969.
- [28] P. Chu and J. Beasley, "A genetic algorithm for the multidimensional knapsack problem," *Journal of Heuristics*, vol. 4, no. 1, pp. 63–86, June 1998.
- [29] D. P. Anderson and G. Fedak, "The computational and storage potential of volunteer computing," in *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, May 16-19 2006, pp. 73–80.