RJ10407 (A0704-025) April 18, 2007
Computer Science

IBM Research Report

SPARK: Integrated Resource Allocation in Virtualization-Enabled SAN Data Centers

Aameek Singh*, Madhukar Korupolu, Bhuvan Bamba*
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

*Georgia Institute of Technology
(work done at IBM Research)

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

SPARK: Integrated Resource Allocation in Virtualization-enabled SAN Data Centers

Aameek Singh[†] **Madhukar Korupolu**[*] **Bhuvan Bamba**[†]

[†]Georgia Institute of Technology
Work done at IBM Research
{aameek, bhuvan}@cc.gatech.edu

[*]IBM Almaden Research Center
Contact Author
madhukar@us.ibm.com

Abstract

In this paper, we present a novel framework called SPARK (Stable-Proposals-And-Resource-Knapsacks) for addressing the combined placement of application data and CPU in virtualized SAN data centers. Unlike previous approaches, which address only CPU placement or only storage placement, SPARK enables both in an integrated manner taking into account the differing degrees of proximity and affinity among the different node pairs in the data center.

SPARK is based on two well-studied problems – Stable Marriage and Knapsacks – and is simple, fast, intuitive, iterative, and versatile. It yields good quality placements – in our synthetic experiments, it is consistently within 4% of the optimal values computed using Linear Programming based methods for a wide range of workloads and experiments. At the same time, it is one to two orders of magnitude faster than LP and scales to much larger problem sizes that arise in practice. In comparison with other natural candidate algorithms, SPARK outperforms them by at least 30-40%, demonstrating faster speed and improved versatility at the same time.

The fast running time of SPARK makes it highly suitable for dealing with dynamic data center complexities: workload surges, growth, node failures and downtimes that need quick response. Its built-in versatility positions it well to accommodate policies and constraints that may arise in practical deployments.

1 Introduction

Server virtualization is becoming a quintessential data center technology. The once ever-expanding and unmanageable data centers are now being tamed by advances in virtual machine (VM) technologies such as VMware [17] and Xen [29]. These allow applications to run in isolated containers without interfering with each other. Applications which run on single machines can now be moved to virtual machines; and these virtual machines in turn can be moved to fewer physical machines thus consolidating hardware, reducing space and management costs, and increasing utilization.

A notable recent development in this domain is the emergence of *live migration* technologies such as VMotion [47] and [34] for Virtual Machines. These enable applications to be moved from one machine to another in real-time without much downtime. This is an extremely useful tool for data center administrators as it facilitates load-balancing, failure-management and system maintenance. Combining these advancements with similar data migration technologies [43, 25, 38, 41] can help create a highly adaptive and dynamic data center environment.

With these advances the prospect of realizing on-demand utility computing in data centers is gaining momentum, both in the industry and in academia [46, 45].

1.1 Recent Trends and Challenges

In spite of the above benefits of server virtualization, managing such virtualized data centers is still a difficult task. Studies estimate that management costs will soon outweigh the hardware costs [7] because of the manual steps and administrator involvement required.

Deciding where to run applications and where to place their data requires planning, especially in response to dynamic data center complexities such as workload surges, growth, node failures and downtimes. Manual planning is slow, error-prone and becomes complex with increasing scale and heterogeneity of data centers and increasing number of workloads and applications. Data centers often go with the safer option of over-provisioning and incur the resulting under-utilization and wastage penalty. Many consider this lack of automated management to be a primary impediment to more wide-spread adoption of virtualization technology in data centers [45].

Fast, autonomic resource planning in modern data centers is a challenging problem. It requires *handling storage and computational resources in a coupled manner*. For example, moving an application VM to another physical node in response to a workload surge needs to take into account the "affinity" of the new node to application's storage. Similar affinities need to be taken into account when moving application storage.

Complexity is introduced into this problem due to the *heterogeneity in data centers*. The hardware and Storage Area Network (SAN) fabric in data centers are incrementally built over time and disparate resources co-exist in the same environment. For example, in the sample data center model shown in figure 1, different switches can be of different capabilities. This results in differing affinities between different node pairs.

Further contributing to this non-uniformity are the *recent trends in SAN hardware*. Vendors are introducing innovative SAN devices to present available specialized processing power at non-traditional locations in the SAN so as to make better use of emerging virtualization technologies. For example:

- The IBM DS8000 storage controllers support POWER5 logical partitioning which makes them suitable for hosting applications [16, 23]. IBM recently demonstrated running a DB2 application in an LPAR to offload OLAP ad-hoc query computation to the storage controller. [39] describes a system that speeds up search of unindexed data by running "searchlets" at the storage system.

- Cisco MDS 9000 switches [20] have multiple Intel x86 processor blades and can be fitted with multiple application modules enabling functionality offload to switches. Cisco has already displayed the feasibility of this idea by implementing *IBM TotalStorage SAN Volume Controller* and *Veritas Storage Foundation for Networks Solution* on the switches.

Such hardware innovations increase the heterogeneity in modern SAN data centers and make available computing resources at varying proximities from storage. The proximity of these nodes to storage if utilized properly can have significant impact on application performance. For example, if a processor on a storage controller is used to run an I/O intensive application and the application's data is on the same controller then the application can access its storage faster, perform better and relieve network resources for other applications.

1.2 State of the Art and Limitations

Much of the prior work in data center resource allocation addresses single resource placement, that is the placement of CPU only or storage only but not both at the same time.

Ergastulum [27], Hippodrome [26] and Minerva [24] attempt to place storage for applications assuming the application CPU locations are fixed. Oceano [28] and Muse [31] provide methods for provisioning CPU resources in a data center but they do not have a mechanism to take into account the underlying storage system and its affinities to CPU nodes. Similarly recent products like VMware Infrastructure-3 [21] and its Dynamic Resource Scheduler (DRS) [22] address monitoring and allocation of CPU resources in a data center. While these are good starting points, they only concentrate on allocation of one resource while ignoring the other.

Similarly, File Allocation problems [35, 30] look at where to allocate files/storage assuming CPU is fixed. Generalized Assignment problems [52, 53] look at assigning tasks to processors (CPUs) but these again are for single resource allocation and not coupled storage and CPU allocation.

Such single-resource allocation approaches cannot take advantage of the virtualization-enabled heterogeneous resources and proximities in modern SAN data centers.

One way to address simultaneous placement of CPU and storage is to model it as a multi-commodity flow problem [10] using one commodity per application and setting up appropriate source and sink nodes for each commodity and connecting them to the underlying resource nodes (more details in §3). However multi-commodity flow problems are known to be very hard to solve in practice even for medium sized instances [42]. Furthermore, they would end up splitting each applications CPU and storage requirements among multiple resource nodes which may not be desirable. And if such splitting is not allowed for some applications, then we would need an unsplittable flow [33] version of multi-commodity flows, which becomes even harder in practice.

Another area of related work is that of co-scheduling data and computation in grid computing [49, 50, 51]. The aim of grid computing is to pool together resources from possibly geographically distributed disparate IT environments over wide area networks [37, 36]. Such environments typically have distributed ownership of resources, so [49, 50] provide protocols for jobs to advertise their needs and resources to advertise their policies and capabilities and provide a mechanism for jobs to approach resources for a potential match. The focus is on schemes to decide policy match when there is decentralized ownership and not on optimization. Such decentralized ownership is not an issue in SAN data centers. [51] builds on [49] to execute data movement and computations efficiently. They attempt to get data and computation on the same grid node but do not attempt to perform any nearness optimization in selection when placement on the same node is not possible. The grid approaches instead employ replication strategies to create replica of the data closer to the compute node which is justifiable in wide-area grid environments. Such replication, however, is not feasible in SAN data center environments.

1.3 Our Contributions

In this paper, we present a novel framework called SPARK (Stable-Proposals-And-Resource-Knapsacks) for addressing some of the above limitations and challenges in virtualized SAN data centers. SPARK provides, to the best of our knowledge, a first known solution for placing application CPU and data simultaneously among resource nodes in a data center.

The SPARK framework automatically favors high I/O intensive applications to go on nearby storage and compute node pairs. It considers heterogeneous resource nodes – host computers, switches, fabrics, storage controllers etc in the SAN data center and treats each one as having a certain capacity of CPU resource and a certain capacity of storage resource, with one being non-zero or both being non-zero or both being zero. For example, a storage controller with spare processing power would have both nonzero, a switch node such as the Cisco MDS 9000 would have nonzero CPU capacity and so on.

It accounts for the differing degrees of proximity and affinity between different node pairs through user-controllable cost or distance functions. Applications are considered as requiring a certain amount of CPU resource and a certain amount of storage resource. Each application may incur a certain cost, for example its throughput multiplied by $distance(u, v)$ or more generally $Cost(A_i, u, v)$, if application A_i's required CPU is placed on node u and storage on node v.

The mechanism for placing CPU and storage simultaneously involves several interesting aspects. For example, if multiple applications compete for the same CPU node or storage node, then the node can pick certain subset of them based on their sizes and values. This is akin to the Knapsack problems [8]. Further, applications need to decide which nodes to compete for – nodes that are easier to get or ones that are most preferred. Here we borrow ideas from the Stable Marriage problem [15]. The latter addresses the question of given a collection of n boys and girls (or students and universities) each with their own ordered list of preferences, can they be paired up so that the marriages are stable in some sense. For details, please see §3.

The SPARK algorithm proceeds in rounds, iteratively improving the CPU and storage placements in each round. A basic paradigm in SPARK involves having applications *propose* to certain candidate nodes for CPU and storage based on their current locations and affinities, and allowing the receiving resource nodes to select a subset of the proposals based on their proposed values and sizes. This is repeated in stages along with a few swap steps to overcome local optima.

A notable feature of SPARK is that it provides high quality placement decisions quickly. In our experiments, we compare SPARK with a Linear Programming based optimal. Due to limits imposed on LP solvers [11], LP solutions could be obtained, and compared against, for small problem sizes only. For a wide range of synthetic workloads and experiments capturing various distributions and topologies, SPARK is consistently **within 4% of the optimal values**.

At the same time it is one to two orders of magnitude faster than LP, solving even large instances in few minutes. This enables SPARK to handle the much larger problem instances that come up in modern data centers which regularly have hundreds to thousands of hosts, controllers and switches and hundreds of applications.

In comparison with other natural candidate algorithms, SPARK consistently outperforms by at least 30-40%, and at the same time is much faster and more versatile. Comparisons in §4 show that SPARK is about an order of magnitude faster than the next best candidate algorithm for large instances.

The other salient feature of SPARK is its **versatility**. As we discuss in §5 this enables it to accommodate various system policies and application preferences that often arise in practical deployments. It can account for an already existing placement configuration and VM and data migration costs and iteratively **improve the current configuration** (ref. §5.1). Given the constantly changing nature of real life data centers and policies, such flexibility is often a necessity in practice.

1.4 Organization

The rest of the paper is organized as follows. In §2, we describe the model SAN environment considered in this work and the architecture of our solution framework. We describe the design of the SPARK algorithm in §3. Detailed experimental evaluation of our algorithm and its comparison with other techniques is presented in §4. Salient features of the SPARK approach and directions for future work are discussed in §5. We finally conclude in §6.

2 System Architecture

In this section, we describe the architecture of our framework and its various components. We start with a description of a model SAN data center environment.

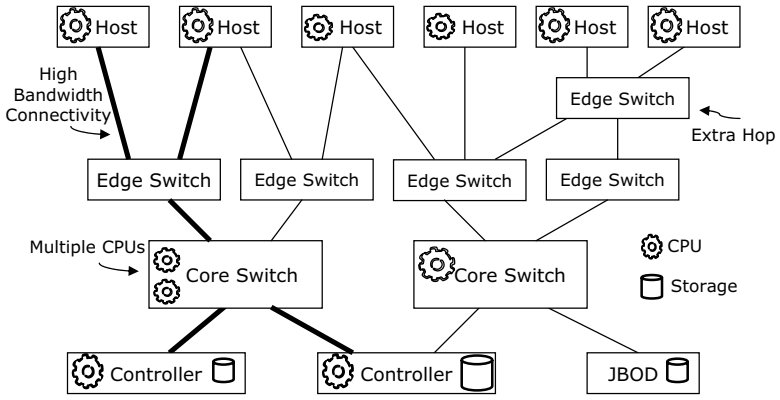

Figure 1: Modern SANs with heterogeneous resources

A storage area network in a data center consists of multiple devices – application servers, edge and core switches, different types of storage devices from high end controllers to Just-a-Bunch-Of-Disks (JBOD). These devices are connected through a high speed network, usually referred to as the *fabric*. Figure-1 shows an example core-edge design SAN topology. As mentioned earlier, modern SANs are heterogeneous in nature with resources differing in bandwidth connectivity, resources capacities and capabilities. Additionally, many non-traditional devices now include significant processing power (Figure-1 shows core switches and storage controllers with CPU nodes). Note that these CPUs are as powerful as ones available in traditional application servers, if not more.

An important characteristic of this modern SAN environment is its support for virtualization platform deployment. Most CPU architectures today have been virtualized including the ones available on storage controllers and switches. For example, Cisco MDS 9000 switches ship with Intel x86 processors, virtualized by VMware, Xen and IBM DS 8000 controllers have PowerPC processors, virtualized through the Logical Partitioning (LPAR) technology [23]. With increasing push towards standardization of VM formats [13], these processors are equal candidates for addition to virtualized resource pools that would normally contain only application servers. In fact, these nodes can be even more effectively utilized when hooks exist for specialized applications that can communicate with fast-path APIs available at these nodes. For example, an application running at the controller storing its database, can significantly improve table scan performance by directly communicating with the controller storage [5].

Such a virtualized environment can be used to realize *utility computing*, where all resources are aggregated into pools and workloads are dynamically mapped to use resources from these common pools. We discuss this goal in the next section.

2.1 Towards Utility Computing

Consider the SAN environment in Figure-1 where a virtualization platform (say, VMware Server [18]) has been installed at all CPU nodes and all applications have been encapsulated into VMware virtual machines. Administrators can choose to run multiple application VMs at each CPU node for efficient resource utilization. Now, suppose an application gets hit with a sudden workload surge and its VM begins using extra CPU resources. This might impact performance of all applications running on that node.

To correct this, the administrator simply moves one of the application VMs on this physical node (*source*) to another VMware server on a *target* node using the VMware migration technology – VMotion [47]. The VM state is encapsulated into regular files and stored in the SAN. The target server accesses these files concurrently and the active memory and execution state of the VM is transmitted over a high speed network. Since the network is also virtualized, the virtual machine retains its network identity and connections, ensuring a seamless migration process. Note that use of VMware technologies is for illustrative purposes only and similar migration technology also exist for Xen [34], which can migrate VMs running *interactive* applications in only tens of milliseconds.

While this technology seemingly brings the utility computing dream within reach, the human involvement in planning for selecting an appropriate VM to move and the appropriate target node for migration can lead to errors, sub-optimal decision making and is also a bottleneck in situations that require immediate response. As mentioned earlier, autonomically performing such coupled placement decisions accounting for storage-CPU affinities, is complex and SPARK aims at addressing this challenge.

2.2 Planner Architecture

In this section, we describe the architecture of our framework detailing its internal components and its interaction with external entities. Figure-2 shows the framework.

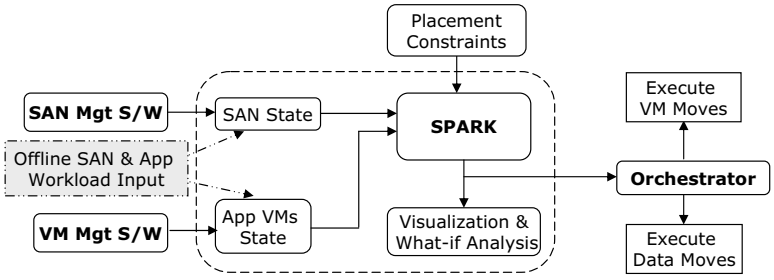

Figure 2: Planner Architecture

External Components: Planning requires detailed information of the underlying storage SAN, which is obtained from *SAN management tools* like EMC Control Center [4] and IBM Total Storage Productivity Center [6]. These tools keep real-time information for all devices and events in the SAN. Additionally, information about the state of application VMs

is required to obtain utilizations and availabilities at processing nodes. This is obtained from *VM management suites* like VMware Infrastructure-3 [21] and similar products [1, 2, 14, 19]. Relevant SAN and VM state information can also be input in an offline mode, especially when doing what-if analysis (see below). *Constraints and preferences* for placing application VMs and storage on certain nodes are also provided as input to the planning framework. Furthermore, the planning engine is connected to an *Orchestrator* that can execute VM and data migration workflows using storage management and VM management tools.

Internal Components: Internally, the framework consists of a *SAN State* component that maintains relevant SAN state required during planning, e.g. SAN topology information. Similarly relevant *VM State* information is maintained. The SPARK algorithm takes this information along with placement constraints to generate a placement plan of application storage and CPU on resource nodes. This plan can be viewed through a *visualization engine* and can also be used as a *what-if analyzer*, through which an administrator can proactively assess the behavior of the system in response to certain dynamic scenarios (e.g. what-if App-A's workload surges by 20%?). Finally, the placement plan is executed through the orchestrator (if desired, the plan can be verified by an administrator). Note that the current focus is primarily on the SPARK algorithm and its performance evaluation while the other pieces such as the visualization engine and what-if analyzer would be part of future work.

Modes of Operation: This framework can be deployed both in an *offline* and an *online* setup. In an offline mode, an administrator can input relevant information (maybe imported from a snapshot of a live system state) and visualize the plans generated by the framework and perform what-if analysis for his/her current setup. In an online mode, the framework can continuously monitor the SAN and VM state to identify workload surges or failures, automatically initiate planning in response and actually execute the plan through the orchestrator.

3 Integrated CPU-Storage Solution

The previous sections have discussed how innovations in virtualization technologies and SAN node specialization have laid the foundation for a *utility computing* platform. An important missing piece in the current systems is the planning scheme that produces placement of workloads to resources maximizing overall performance. Such a scheme has to be fast, scalable and robust to handle a wide variety of workloads and SAN environments. In this section, we describe a first step in this direction through SPARK and start with the mathematical problem formulation.

3.1 Problem Formulation

A SAN in the data center has multiple types of nodes – application servers, switches, storage controllers – with available CPU and storage resources. Let the set of CPU resource nodes in the data center be denoted by $\mathcal{P} = \{P_k : 1 \leq k \leq |\mathcal{P}|\}$. This includes all CPU resources in the SAN where an application can be run – e.g., an application server, a switch with a virtualizable CPU or a high-end storage controller. Each such CPU resource P_k has an associated limit on how much processing capacity it has, say in processor cycles per second, which we denote by $cap(P_k)$.

Similarly let the set of storage resource nodes in the data center be denoted by $\mathcal{S} = \{S_j : 1 \leq j \leq |\mathcal{S}|\}$. Each such storage resource S_j has an associated limit on how much storage capacity it has, say in GBs, denoted by $cap(S_j)$. These resources are connected to each other directly or indirectly based on the SAN topology (e.g. Figure-1).

Along with these, there is a set of applications, say $\mathcal{A} = \{A_i : 1 \leq i \leq |\mathcal{A}|\}$ that need to run in the data center. These applications are packaged into virtual containers and can be executed on compatible virtualization platforms deployed at CPU nodes in the SAN. Each such application VM has a CPU resource requirement and a storage resource requirement. We use $A_i.StgReq$, $A_i.CPUReq$, and $A_i.Thpt$ to denote the amount of storage required (GB), the amount of CPU required (processor cycles per second), and the I/O throughput between the storage and CPU (post-cache IOs per second) for the application A_i. Also, $A_i.Stg$ and $A_i.CPU$ denote the storage and CPU nodes that host application data and VM respectively.

Cost Function: Given these, the question is where to allocate CPU and storage for each application, that is where should each application be run among the available CPU resources and where should its data be placed among available storage resources. For each application, we use $\boldsymbol{Cost(A_i, S_j, P_k)}$ to denote the cost of running application A_i on storage resource node S_j and CPU resource node P_k. We also refer to it as the "*distance*" between S_j and P_k for A_i.

This cost function is used to capture the affinities, or lack thereof, between various application, storage and CPU groups. For example, if an application has preference for storage nodes of a certain type (e.g., RAID5) then the costs of assigning it to that storage node could be set lower. Also, if an application has a hard latency requirements then all storage-CPU pairs that cannot satisfy the latency bound have their cost set to infinity. In general, the cost function gives us a flexible way to account for multiple environmental characteristics like SAN fabric design, application QoS requirements, application preferences for specialized nodes or even the dollar cost of storage and processors (for example, biasing cheaper storage).

In our current design, the cost function is designed by the SAN administrator to achieve their specific objectives (we describe one example cost function below). It is to be noted however that many of the SAN characteristics that influence the cost function can be obtained automatically from the underlying SAN management tools and policy databases (e.g. fabric design, end-to-end latencies). The SPARK algorithm and the framework presented below are flexible enough to work with any cost function provided as input. This makes SPARK extremely versatile in dealing with many peculiarities of a real SAN data center environment (see §5.1).

Given this cost function for all applications and resources, the goal of the placement is to select locations for the CPU and storage of each application so as to keep the overall cost for all applications small, that is,

$$min \sum_i Cost(A_i, A_i.Stg, A_i.CPU) \qquad (I)$$

while ensuring feasibility by not exceeding storage and CPU resource node capacities:

$$\forall j \sum_{i:A_i.Stg=S_j} A_i.StgReq \leq cap(S_j)$$

$$\forall k \sum_{i:A_i.CPU=P_k} A_i.CPUReq \leq cap(P_k)$$

As mentioned above, SPARK can work with any cost function, but for concreteness in this paper we use a cost function that captures desired features and complexity and yet is easy to describe. It tends to keep applications with high throughput requirement on storage-CPU pairs with small distances thus lowering the traffic on the SAN and increasing room for surges and other traffic. It is based on the throughput of the application and the distance between its CPU and storage nodes. It is given by $A_i.Thpt * dist(A_i.Stg, A_i.CPU)$, where latter is the physical inter-hop distance between $A_i.Stg$ and $A_i.CPU$ and is independent of applications. This function notably captures some of the desired features discussed earlier. For example, if a CPU at a storage controller is available, it would have low distance between its storage and CPU node, and thus an application with high I/O throughput would be favored to go there, which improves performance and reduces the load on the SAN network.

Problem Complexity: This problem (I) captures the basic questions inherent in placing CPU and storage in a coupled manner. The NP-Hard nature of the problem can be established by reducing to the 0/1 Knapsack problem. Even if a simpler case of our problem – involving two CPU nodes (one of which is a catch-all node of infinite capacity and large cost) and fixed application storage – can be solved, it can be used to solve the Knapsack problem by making the second CPU node correspond to the knapsack and setting the costs and CPU requirements accordingly. Having to decide coupled placements for both storage and CPU with general cost/distance functions makes the problem more complex.

Special case of one resource fixed: If either CPU or storage locations are fixed and only the other needs to be determined then there is related work as mentioned in §1 along the lines of (a) File Allocation Problem [30, 35] placing files/storage assuming CPU is fixed; (b) Minerva, Hippodrome [24, 26] assume CPU locations are fixed while planning storage placement; (c) Generalized Assignment problems [52, 53]: Assigning tasks to processors (CPUs).

Modeling as flow problems: If the storage and CPU requirement for applications can always be split across multiple resource nodes, then one could also model it as a multi-commodity flow problem [10] – one commodity per application, introduce a source node for the CPU requirement of each application and a sink node for the storage requirement, with the source node connected to CPU resource nodes, storage resource nodes connected to the sink node and appropriate costs on the storage-CPU resource node pairs. However multi-commodity flow problems are known to be very hard to solve [42] in practice even for medium sized instances. And if the splitting is not justifiable for applications (e.g. it requires sequential processing at a single server), then we would need an unsplittable flow [33] version for multi-commodity flows, which becomes even harder in practice.

Special case of uniform costs. Another important aspect of the problem is non-uniform costs. In a modern virtualized data center these costs vary depending on various factors like application preferences, storage costs, node heterogeneity and distances. If these variations were not present, i.e. costs for each application A_i were the same for all (S_j,P_k) pairs then the problem could be simplified to placing storage and CPU independently without coupling. In the next section, we discuss an algorithm INDV-GR that follows this approach. The evaluation and discussion of its performance in the general data center environment is given in the experimental section.

3.2 Algorithm

In this section, we begin by outlining two simpler algorithms – a greedy individual placement algorithm INDV-GR that places CPU and storage of applications independently in a natural greedy fashion and a greedy pairs placement algorithm PAIR-GR that considers applications in a greedy fashion and places each ones CPU, storage pair simultaneously. The pseudocode for these is given as Algorithm-1 and Algorithm-2.

The INDV-GR algorithm (Algorithm-1) first places application storage by sorting applications by $\frac{Thpt}{App.StgReq}$ and greedily assigning them to storage nodes sorted by $BestDist$, which is the distance from the *closest* CPU node. Intuitively, INDV-GR tries to place highest throughput applications (normalized by their storage requirement) on storage nodes that have the closest CPU nodes. In the next phase, it will similarly place application VMs on CPU nodes.

Algorithm 1 INDV-GR: Greedy Individual Placement

```
1:  RankedAppsStgQ ← Apps sorted by Thpt/StgReq // decr.
2:  RankedStgQ ← Storage sorted by BestDist // incr.
3:  while RankedAppsStgQ ≠ Ø do
4:      App ← RankedAppsStgQ.pop()
5:      for (i=0; i<RankedStgQ.size; i++) do
6:          Stg ← RankedStgQ[i]
7:          if (App.StgReq ≤ Stg.AvlSpace) then
8:              Place App storage on Stg
9:              break
10:         end if
11:     end for
12:     if (App not placed) then
13:         Error: No placement found
14:     end if
15: end while
16: Similar for CPU placement
```

However, due to its greedy nature, a poor placement of application can result. For example, it can place an application with 600 units storage requirement, 1200 units throughput at a preferred storage node with capacity 800 units instead of choosing two applications with 500 and 300 units storage requirement and 900, 500 units throughput (cumulative throughput of 1400). Also, INDV-GR does not account for storage-CPU affinities beyond using a rough $BestDist$ metric. For example, if A_i storage is placed on S_j, INDV-GR does not especially try to place A_i CPU on the node closest to S_j.

This can potentially be improved by a greedy simultaneous placement. The PAIR-GR algorithm (Algorithm-2) attempts such a placement.

It tries to place applications sorted by $\frac{Thpt}{CPUReq*StgReq}$ on storage, CPU **pairs** sorted by the distance between the nodes of the pair. With this, applications are placed simultaneously into storage and CPU buckets based on their affinity measured by the distance metric.

Notice that PAIR-GR also suffers from the shortcomings of the greedy placement where an early sub-optimum decision results in poor placement. Ideally, each storage (and CPU) node should be able to select application *combinations*

Algorithm 2 PAIR-GR: Greedy Pairs Placement

1: $RankedAppsQ \leftarrow$ Apps sorted by $\frac{Thpt}{CPUReq*StgReq}$
2: $RankedPairsQ \leftarrow$ {Storage x CPU} sorted by distance
3: **while** $RankedAppsQ \neq$ Ø **do**
4: $App \leftarrow RankedAppsQ$.pop()
5: **for** (i=0; i<RankedPairsQ.size; i++) **do**
6: $Stg \leftarrow RankedPairsQ[i].storage()$
7: $CPU \leftarrow RankedPairsQ[i].CPU()$
8: **if** ($App.StgReq \leq Stg.AvlSpace$ AND $App.CPUReq \leq CPU.AvlCPU$) **then**
9: Place App storage on Stg, App CPU on CPU
10: $break$
11: **end if**
12: **end for**
13: **if** (App not placed) **then**
14: *Error: No placement found*
15: **end if**
16: **end while**

that best minimize the overall cost value of the system. This hints at usage of **Knapsack**-like algorithms [8]. Secondly, an important missing component of these greedy algorithms is the fact that while applications have a certain preference order of resource nodes they would like to be placed on (based on the cost function), the resource nodes would have a different preference determined by their capacity and which application combinations fit the best. Matching these two distinct preference order indicates a connection to the **Stable-Marriage** problem [15] described below.

3.3 The SPARK algorithm

The above discussion about the greedy algorithms suggested an intuitive connection to the Knapsack and Stable Marriage problems. These form the basis for the design of SPARK. So we begin by a brief introduction of these problems.

Knapsack Problem[8, 48]: Given n items, a_1 through a_n, each item a_i has size s_j and a profit value v_j. The total size of the knapsack is S. The 0-1 knapsack problem asks for the collection of items to place in the knapsack so as to maximize the profit. Mathematically:

$$max \sum_{j=1}^{n} v_j x_j \;\; subject\ to \;\; \sum_{j=1}^{n} s_j x_j \leq S$$

where x_j = 0 or 1 indicating whether item a_j is selected or not. This problem is known to be NP-Hard and has been well-studied for heuristics of near optimal practical solutions [40].

Stable Marriage Problem [15, 44]: Given n men and n women, where each person has a ranked preference list of the members of the opposite group, pair the men and women such that there are no two people of opposite group who would both rather have each other than their current partners. If there are no such people, then the marriages are said to be "stable". This is similar to the residency-matching problem for medical graduate applicants where each applicant submits his ranked list of preferred medical universities and each university submits its ranked list of preferred applicants.

The Gale-Shapely *Proposal algorithm* [44] is the one that is commonly used in such problems. It involves a number

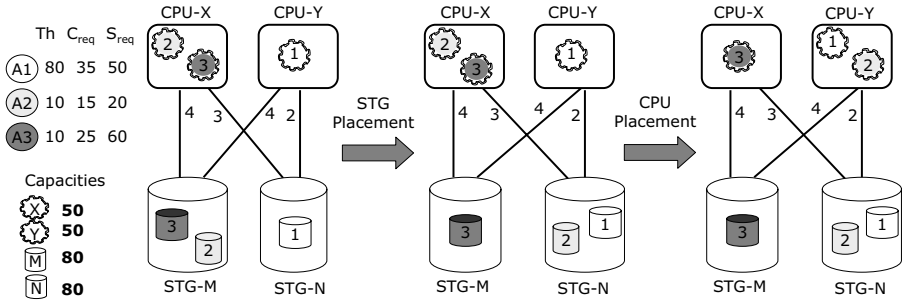

Figure 3: Placement in rounds. STG placement brings A_2-storage closer to A_2-CPU and CPU placement further improves by bringing A_2-CPU closer to A_2-storage. Knapsacks help choose A_1+A_2 combination over A_3 during placement.

of "rounds" (or iterations) where each man who is not yet engaged "proposes" to the next most-preferred woman in his ordered list. She then compares the proposal with the best one she has so far and accepts it if it is higher than her current one and rejects otherwise. The man who is rejected becomes unengaged and moves to the next in his preference list. This iterative process is proved to yield stable results [44].

Notice that placing an application together on storage, CPU resource pair (S_j, P_k) (as done by PAIR-GR) would impact resource availability in all overlapping pairs (S_j, P_l) and (S_m, P_k). This *overlap* can have cascading consequences. This indicates that perhaps placing storage and CPU separately, yet coupled through affinities would hold the key to solving this problem. Combining this observation with knapsacks and the stable proposal algorithm leads us to SPARK.

SPARK: Consider a general scenario where say the CPU part of applications has been placed and we have to find appropriate locations for storage. Each application A_i first constructs an ordered preference list of storage resource nodes as follows: Let P_k be the processor node where the CPU of A_i is currently placed. Then all S_j, $1 \leq j \leq \mathcal{S}$ are ranked in increasing order of $Cost(A_i, S_j, P_k)$, or with our example cost funcation, $A_i.Thpt * dist(S_j, P_k)$[1]. Once the preference lists are computed, each application begins by proposing to the first storage node on its list (like in the stable-marriage scenario). On the receiving end, each storage node looks at all the proposals it received. It computes a profit value for each such proposal that measures the utility of that proposal. How to compute these profit values is discussed in §3.4. We pick the storage node that received the highest cumulative profit value in proposals and do a knapsack computation for that node [2]. This computation decides the set of applications to choose so as to maximize the total value without violating the capacity constraints at the storage resource. These chosen applications are considered accepted at the storage node. The other ones are rejected. The rejected ones move down their list and propose to the next candidate. This process repeats until all applications are accepted. The pseudocode for this part is given in Algorithm-3.

We assume a dummy storage node S_{dummy} (and similarly a dummy CPU node P_{dummy}) of unlimited capacity and large distance from other nodes. These would appear at the end of each preference list ensuring that the application would be accepted somewhere in the algorithm. This catch-all node provides a graceful termination mechanism for the algorithm.

Given these storage placements, the algorithm then decides the CPU placements for applications based on the affinities from the chosen storage locations. The pseudocode for the CPU part is similar to the one in Algorithm-3.

[1] In case CPUs have not been placed (for example, when doing a fresh placement) we use the $BestDist$ metric.

[2] Though knapsack problem is NP-Hard, there are known polynomial time approximation schemes (PTAS) for it [32] which work reasonably well in practice to give not exactly optimal but close to optimal solutions. We use one such package [9] here.

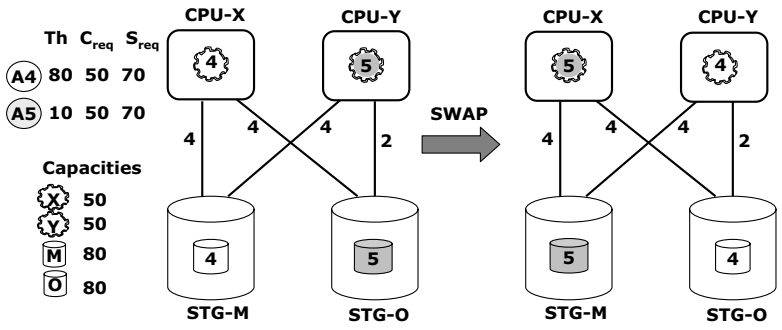

Figure 4: Swap exchanges STG/CPU pairs between A_4, A_5. Individual rounds cannot make this move - during STG placement, M and O are equally preferable for A_1 as they have the same distance from A_4-CPU (X). Similarly during CPU placement.

An illustration for the working of the SPARK-Stg and SPARK-CPU rounds is given in Figure 3. SPARK-Stg brings $A_2.Stg$ closer to $A_2.CPU$ and SPARK-CPU further improves by bringing $A_2.CPU$ closer to $A_2.Stg$. Knapsacks help choose the $A_1 + A_2$ combination over A_3 during placement.

Though the combination of SPARK-Stg and SPARK-CPU address many possibilities well, they are not equipped to deal with scenarios like the one shown in Figure- 4. Here a move of one end during a round of placement (either storage or CPU) doesnt improve the placement but moving of both simultaneously does. This is where the SPARK-Swap step comes in. It takes two applications A_i and $A_{i'}$ and exchanges their CPU and storage locations if that improves the cost while still being within the capacity limits.

Combining these insights, the SPARK algorithm is summarized in Algorithm-4. It proceeds iteratively in rounds. In each round it does a proposal-and-knapsack scheme for storage, a similar one for CPU, followed by a Swap step. It thus improves the solution iteratively, until a chosen termination criterion is met or until a local optimum is reached.

3.4 Computing Profit Values

One of the key steps in the SPARK algorithm is how to compute the profit values for the proposals. Recall that when a storage node S_j receives a proposal from an application A_i it first determines a profit value for that proposal which it then uses in the knapsack step to determine which ones to accept.

We distinguish two cases here based on whether A_i currently has a storage location or not (for example, if it got kicked out of its location, or it has not found a location yet). If it does, say at node $S_{j'}$ ($S_{j'}$ must be below S_j in A_i's preference list, otherwise A_i wouldnt have proposed to S_j.) Then the receiving node S_j would look at how much the system would save in cost if it were to accept A_i. This is essentially $Cost(A_i, S_{j'}, P_k) - Cost(A_i, S_j, P_k)$ where P_k is the current (fixed for this storage placement round) location of A_i's CPU. This is taken as the profit value for A_i's proposal to S_j.

On the other hand, if A_i does not have any storage location or if A_i has storage at S_j itself, then the receiving node S_j would like to see how much more the system would lose if it did not select A_i. If it knew which storage node $S_{j'}$, A_i would end up not selected it then the computation is obvious. Just taking a difference as above from $S_{j'}$ would give the necessary profit value. However where in its preference list A_i would end up if S_j rejects it, is not known at this time.

In the absence of this knowledge, a conservative approach is to assume that if S_j rejects A_i, then A_i would go all the way to the dummy node for its storage. So with this, the profit value can be set to $Cost(A_i, S_{dummy}, P_k) -$

Algorithm 3 SPARK-Stg: Storage placement in SPARK

for all App **in** $AppsQ$ **do**
  Create storage preference list sorted by distance from current App CPU
  Propose to best storage
end for
while (All apps not placed) **do**
  $MaxStg \leftarrow StgQ[0]$
  for all Stg **in** $StgQ$ **do**
    Compute proposals profit
    $MaxStg \leftarrow max(MaxStg.profit\,, Stg.profit)$
  end for
  Knapsack $MaxStg$
  for all Accepted apps **do**
    Place App stg on $MaxStg$
  end for
  for all Rejected apps **do**
    Propose to next storage in preference list
  end for
end while

Algorithm 4 SPARK: Proposals-and-Knapsacks

MinCost $\leftarrow \infty$, SolutionCfg $\leftarrow$ Ø
loop
  SPARK-Stg()
  SPARK-CPU()
  SPARK-Swap()
  Cost $\leftarrow 0$
  for all App **in** $AppsQ$ **do**
    Cost $\leftarrow$ Cost + App.Thpt * dist(App.Stg, App.CPU)
  end for
  if (Cost $<$ MinCost) **then**
    MinCost $\leftarrow$ Cost
    SolutionCfg $\leftarrow$ current placement solution
  else
    $break$ // termination by local optimum
  end if
end loop
return SolutionCfg

$Cost(A_i, S_j, P_k)$.

An aggressive approach is to assume that A_i would get selected at the very next storage node in its preference list after S_j. In this approach, the profit value would then become $Cost(A_i, S_{j'}, P_k) - Cost(A_i, S_j, P_k)$ where $S_{j'}$ is the node immediately after S_j in the preference list for A_i. The reason this is aggressive is that $S_{j'}$ may not take A_i either because

it has low capacity or it has much better candidates to pick.

In this paper we work with the conservative approach described above. Experiments show that the solutions computed by the SPARK algorithm are very close (within 4%) to the optimal with this approach for a range of scenarios. In future, we plan to examine sophisticated approaches including estimating probabilities that a given item would be accepted at a particular node based on history from past selections.

4 Experimental Evaluation

In this section, through a series of simulation based experiments, we evaluate the performance of SPARK for various workload and SAN environments. We also compare it with other candidate algorithms and Linear Programming based optimal solutions. Simulation based experiments allow us to evaluate SPARK for a wide range of environments, something which is tough to do in real data center systems. However, as part for our future work we are also collecting datasets from real operational systems for a similar evaluation. §4.1 describes the experimental setup followed by the results. We summarize our findings in §4.6.

4.1 Setup

To evaluate SPARK, we simulated storage area networks of varying sizes and application workloads with different CPU, storage requirements and I/O throughput rates.

As in any realistic SAN environment, the size of the SAN is based on the size of the application workload. We used simple ratios for obtaining the number of application servers, controllers and switches. For example, for a workload with 1000 applications (one CPU, storage node per application), we used 333 (Ratio=3) application servers, 50 (Ratio=20) high end storage controllers (with CPUs) and 40 (Ratio=25) regular storage devices (without CPUs). We used the established industry best-practices based core-edge design of the SAN with storage controllers connected to application servers through three levels of core and edge switches. For the above example, we had 111 edge switches, 16 mid level core switches and 5 core switches (with CPUs). We used uniform CPU and storage capacities for nodes.

All these parameters are encapsulated into a single metric called **Problem Size** which is equal to the product of number of applications, number of CPU nodes and number of storage nodes. It roughly represents the complexity of the problem.

The CPU and storage requirements for applications are generated through a normal distribution with varying mean and standard deviation. To avoid using a large number of parameters, we used the same mean and std-dev for these distributions (independently). The I/O throughput rates were also varied using a normal distribution. We describe the exact mechanism used to obtain these values in §4.1.2 and evaluate our algorithm with different values in §4.3 and §4.4.

Another important input is the application CPU-storage distance matrix. In our experiments, we used same distance values independent of applications derived using an exponential function based on the physical inter-hop distance; the closest CPU-storage pairs (both nodes inside a high end storage controller) are set at distance 1 and for every subsequent level (core switch CPU and storage and then hosts and storage) the distance value is multiplied by a *distance-factor*. We present experiments with varying distance factor in §4.5.

4.1.1 Algorithms and Implementations

We compared the following algorithms in our evaluation. All algorithms were implemented in C++ and run on a Windows XP Pro machine with Pentium (M) 1.8 GHz processor and 512 MB RAM. For experiments involving time, results were averaged over multiple runs.

- **Individual Greedy Placement (INDV-GR)**: The greedy algorithm that independently places application storage and CPUs as shown in Algorithm-1 (ref. §3).
- **Pairwise Greedy Placement (PAIR-GR)**: The greedy algorithm that places applications into best available storage-CPU pairs – Algorithm-2 (ref. §3).
- **OPT-LP**: The optimal solution obtained by the LP formulation. We used CPLEX Student [3] for obtaining integer solutions (it worked only for the smallest problem size) and popular MINOS [12] solver (through NEOS [11] web service) for fractional solutions in the [0,1] range for other sizes. We could only test upto 300 application nodes (problem size = 1.1 M) as the number of variables grew past the limits of the solvers after that.
- SPARK: Our SPARK algorithm as described in §3. It used the 0/1 knapsack algorithm contained in [48] with source code available from [9].
- SPARK-**R1**: The solution obtained after only a single round of SPARK. This helps illustrate the iteratively improving nature of SPARK.

4.1.2 Design of Experiments

We conducted the following set of experiments to evaluate the above algorithms on a wide range of workload distributions, problem sizes, and SAN topologies:

– Scalability Tests: An important requirement for SPARK is to be able to handle large data center environments. For validating this, we measure its performance with increasing size of the storage area network. As mentioned earlier, size of the SAN is computed based on the total number of applications in the workload. We varied this number from 10 (problem size 140) upto 2500 (problem size of 575 M). Please note that the OPT-LP implementation could only solve till 300 nodes (problem size 1.07 M). We report these experiments in §4.2.

– Varying Mean: Any fitting algorithm would be influenced by the required "tightness" of the fit. We varied the mean of the normal distributions used to generate workload CPU and storage requirements. For CPU requirements, the mean (μ) is varied through a parameter $\alpha \in (0, 1]$ using the formula $\mu = \frac{\alpha * CPU_{cap}}{N}$, where CPU_{cap} is the capacity of the CPUs and N is the ratio of number of applications to available CPU nodes (similarly for storage requirements). It is easy to see that $\alpha = 1$ implies the strictest packing arrangement where the average CPU requirement is equal to the CPU_{cap} divided by the number of applications per CPU. Please note that high α values might not have any feasible solution for any solver.

– Varying Std-dev: Along with fitting tightness, uniformity of the workloads will also play an important role in the performance of various fitting algorithms. To evaluate this, we varied the standard deviation of the normal distribution as

a function of $\beta \in [0, 1]$ using the formula $\sigma = \frac{\beta * CPU_{cap}}{N}$ where parameters are defined as above.

– Varying Distance Factor (DF): DF determines the distances between storage nodes and CPU nodes at various levels. A higher distance factor implies a greater relative distance between two levels of CPU nodes from the underlying storage nodes. For example, if a CPU is accessing storage through a wide area network, its distance from the storage is much higher than any CPU accessing through the SAN fabric. This set of experiments provides interesting insights into placement characteristics of various algorithms (ref. §4.5).

4.2 Scalability Test: Varying SAN Size

In this experiment, we increased the size of the SAN with increasing number of applications in the workload. The problem size varied from 140 to 575M representing a small 10 applications workload increasing upto 2500. The other parameters were α=0.55, β=0.55, DF=2. We measured the quality of the optimization and solution processing time for all implementations. Recall that quality is measured using the cost metric given by cumulative sum of the application throughput times its storage-CPU pair distance.

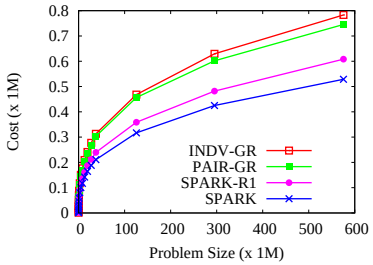

Figure 5: Quality with varying size. *OPT-LP only works upto 1.1 M size. See Fig-6.*

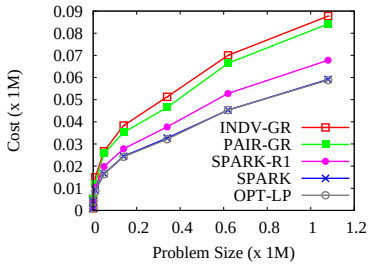

Figure 6: Comparison with OPT-LP. SPARK *is within 0-4%. See Table-1*

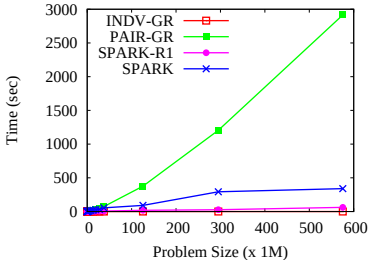

Figure 7: Time with varying size. *Even for 1M size, OPT-LP took 30+ minutes.*

Figure-5 shows the quality of optimization for INDV-GR, PAIR-GR, SPARK, SPARK-R1. Since OPT-LP only works upto a problem size of 1.1 M (as the number of variables in the LP formulation go past the limits of the solvers [11]), we show a zoomed graph for small sizes in Figure-6 and give exact cost measures in Table-1.

First notice the separation between the greedy algorithms and SPARK in Figure-5. Of the two greedy algorithms, PAIR-GR does better as it places applications based on storage-CPU pair distances, whereas INDV-GR's independent

decisions do not capture that. It is interesting to see that SPARK-R1 performs better than both greedy algorithms. This is because of using knapsacks (thus picking the right application combinations to place on resources) and placing CPU based on the corresponding storage placement. With every subsequent round, SPARK iteratively improves to the best value shown.

Figure-6 shows the quality of all algorithms including OPT-LP for small problem sizes. While rest of the trends are similar, it is most interesting to see the closeness in curves of OPT-LP and SPARK. Table-1 shows the exact values of OPT-LP and SPARK optimization quality and SPARK is within 2% for all but one case where its within 4% of OPT-LP. This validates excellent optimization quality of SPARK.

Size	SPARK-**R1**	SPARK	**OPT-LP**	**Difference**
0.00 M	986	986	986	0%
0.01 M	10395	9079	8752	3.7%
0.14 M	27842	24474	24200	1.1%
0.34 M	37694	32648	32152	1.5%
0.62 M	52796	45316	45242	0.1%
1.08 M	67805	59141	58860	0.4%
2.51 M	91717	79933	–	–
4.84 M	114991	100256	–	–
8.27 M	136328	117118	–	–

Table 1: Comparison with OPT-LP (*OPT-LP works only upto 1.08 M due to solver [12] being unable to handle large number of variables and constraints*)

Figure-7 shows the time taken by implementations in giving a solution. As expected, INDV-GR is extremely fast since it independently places application CPUs and storage, thus complexity of $|Apps| * (|STG| + |CPU|)$. On the other hand PAIR-GR first generates all storage-CPU pairs and places applications on pairs giving it a complexity of $|Apps| * |STG| * |CPU|$. Each SPARK round would place storage and CPU independently. SPARK-R1 curve shows the time for the first round which is very small. The total processing time in SPARK would be based on the total number of rounds and complexity of the knapsacks in each round. As shown in the graph, it is extremely competitive taking only 333 seconds even for the largest size of the problem (575 M). Because of its iterative nature and reasonable quality of SPARK-R1, SPARK can actually be prematurely terminated if a quicker decision is required and thus still provide a better solution than comparable algorithms.

4.3 Varying Mean

Our next experiments evaluate the algorithms with varying mean of the normal distribution used to generate CPU and storage requirements for the applications. As mentioned earlier, this mean is varied using the α parameter; higher α increases *tightness* of the fit by increasing the mean.

Figure-8 shows the results with α from 0.1 to 0.7 for a problem size of 37 M (1000 applications), $\beta = 0.55$ and $DF = 2$. The costs of all algorithms increase with increasing α. This is expected since tighter fittings will have fewer combinations that fit, resulting in overall increased cost.

Of the greedy algorithms, PAIR-GR worsens at a faster rate which is due to increasing impact of the *overlap* effect

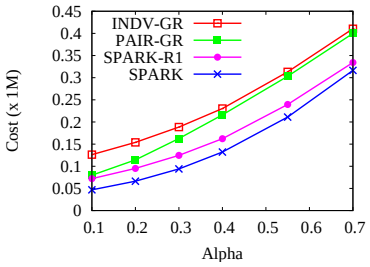

Figure 8: Quality with varying mean

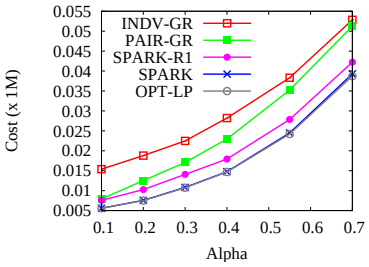

Figure 9: Comparison with OPT-LP

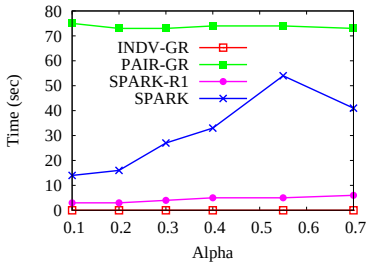

Figure 10: Time with varying mean. SPARK *used 3, 3, 5, 5, 7, 6 rounds*.

as mentioned in §3. Comparatively, SPARK continues to outperform other algorithms for all α values. Also, as earlier, we plotted a similar graph for a smaller problem size of 0.14M (150 applications) to compare with OPT-LP in Figure-9. SPARK remains close to OPT-LP throughout, validating its power to find solutions with varying tightness of the fitting.

For processing time (Figure-10), the greedy algorithms remain constant with increasing α values as they only traverse storage and CPU lists per application. On the other hand, SPARK might need additional rounds of placements in order to converge to a local optimum value. As seen in the graph, while SPARK-R1 remains small, the total time increases overall due to increased number of rounds (varies between 3 and 7 in these experiments). Note that due to convergence to a *local* optimum, there would not be a consistent pattern with increasing α. The objective of the time experiments is to ensure that varying fitting-tightness does not significantly deteriorate SPARK in quality or processing time.

4.4 Varying Std-Dev

Similar to tightness of the fit, a relevant parameter is the uniformity of the workloads. For example, if all workloads have same CPU and storage requirements and same throughout rates, all solutions tend to have the same cost metric (as not much can be done with different combinations of applications during placements). As workloads become less uniform, there are different fittings that might be possible and different implementations will react differently.

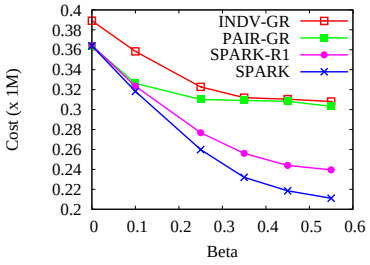

Figure 11: Quality vs std-dev

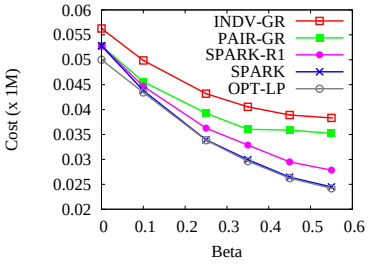

Figure 12: Comparison with OPT-LP

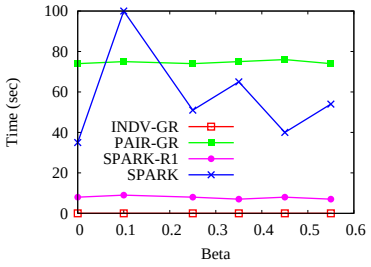

Figure 13: Time with varying std-dev. SPARK *used 3, 10, 6, 8, 5, 7 rounds*.

Figure-11 shows the performance of the algorithms when β is varied from 0 to 0.55 for a problem size of 37 M (1000 applications workload), α=0.55 and DF=2. Notice that as β and thus non-uniformity of workloads increases, costs drop for all algorithms. This is because with availability of many applications with low resource requirements, it is possible to fit more applications at smaller distances. However, SPARK reacts the best to this and significantly drops in cost for higher β values. This is explained through the knapsacks component of SPARK as in case of greedy algorithms, an early

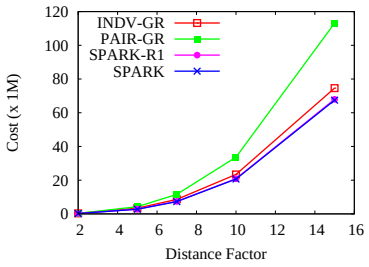

Figure 14: Quality vs distance factor. *INDV-GR fits better at higher DFs.*

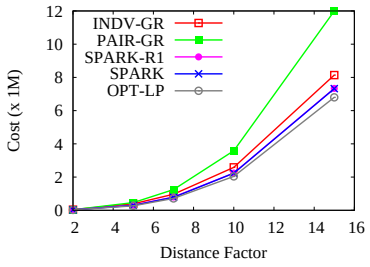

Figure 15: Comparison with OPT-LP. SPARK *is still within 8% of OPT-LP at DF=15.*

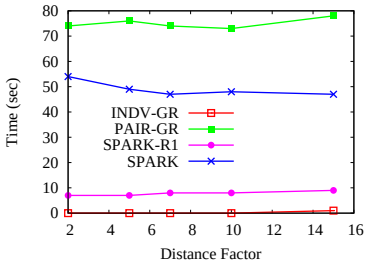

Figure 16: Time vs distance factor

sub-optimum decision can cause poor fitting (e.g. choosing 6 before 5 & 3 were available for an 8 capacity resource).

Figure-12 shows the graph with OPT-LP for smaller problem size of 0.14M (150 applications). Once again, SPARK is able to maintain its effectiveness in comparison to OPT-LP. This means that SPARK adjusts superbly with various workload characteristics. The times of different algorithms for the larger 37M problem with varying β are shown in Figure-13. As mentioned earlier, greedy algorithms tend to maintain constant time based on application and storage-CPU lists. In contrast, total time in SPARK is impacted by the number of rounds required to converge to the local optimal and does not have a consistent pattern. In this case, it used 3, 10, 6, 8, 5 and 7 rounds. However, it is important to note that the total time is still small and does not compromise on SPARK's application to addressing dynamic SAN events like workload surges.

4.5 Varying Distance Factor

The last set of experiments vary the distance factor (DF) that is used to obtain distance values between CPU and storage nodes. As mentioned earlier, we used uniform distance values across applications (that is, distance between a CPU and storage node is same for all applications) and the distance value was obtained using the formula DF^p where p is the physical inter-hop distance between nodes. A higher DF value implies that distance values increase much more rapidly with every hop, for example, due to increased latencies at switches, or going over a LAN or WAN for farther nodes.

The objective of this experiment is to better understand the characteristics of the algorithms as differences in fitting arrangements would be amplified at higher DF values (fitting an application at distance i vs. fitting it at distance j has the cost difference of $Thpt * (DF^j - DF^i)$).

Figure-14 plots the graph as DF is varied from 2 to 15 for a problem size of 37 M (1000 applications) and α=β=0.55 (due to small difference between SPARK and SPARK-R1 relative to the scale of the graph, they appear as if a single line). The most interesting, and in fact surprising observation is the performance of INDV-GR. Not only does it outperform PAIR-GR at higher DFs but it comes reasonably close to SPARK. On close inspection, we found that INDV-GR was performing remarkably at placing applications at higher distances, that is, even though it fit fewer applications (and throughput) at lower distances (p=0 and p=1), it outperformed other algorithms by fitting more at p=3 and less at p=4. It requires further analysis to fully explain this characteristic[3], but we believe it can hint at improving SPARK performance for higher DFs as well. For a smaller problem size of 0.14 M, shown in Figure-15, the OPT-LP is able to separate itself

[3]It is impacted by how applications are fit at lower distances, as only the ones that do not fit there are candidates for fitting at higher distances

from SPARK at higher DF values, though SPARK still remains within 8%. This is due to the amplification as well. Finally, Figure-16 shows that the processing time does not vary much with changing DF indicating it does not impact the fitting.

4.6 Summary of Results

Below, we summarize our key findings:

- SPARK is **scalable** in optimization quality and processing time. With increasing size of the SAN and workloads, SPARK continues to perform (30-40%) better than other candidate algorithms and is **within 4% of LP based solutions** (tested for smaller problem sizes). It is also very fast to compute results taking 5.5 minutes for the largest instance with 2500 applications.
- The **iterative** nature of SPARK and its good performance even after a single round (SPARK-R1 in graphs) allows improving any initial configuration, with an attractive property of trading off time with quality.
- SPARK is **robust** with changing workload characteristics like tightness of the fit and uniformity of workloads. For changing α, β and distance factor parameters, it maintains superior performance over other algorithms and closeness to the optimal.

5 Discussion

SPARK promises to be a good framework for planning resource allocation in modern data center environments. It has many attractive properties that make it a building block for future work in this area. Below we discuss some of its important characteristics and possible directions of research that it can spawn.

5.1 Salient Features

1. Being iterative: The SPARK algorithm proceeds in rounds iteratively improving the solution, thus offering a natural time-quality tradeoff if one is desired for time-constrained domains. The experimental section shows that even after one round SPARK yields better results than other greedy algorithms and multiple rounds further improve the value. For example, for the largest problem size, SPARK-R1 is nearly 20% better than the greedy algorithms with almost the same speed as the fastest INDV-GR.

2. Flexible cost measure for (Application, Storage, CPU**) triple:** The algorithm and the framework allow the cost for each triple to be set independently. At any internal step, say if we have fixed a storage location S_j for A_i and trying to look for better candidate locations for its CPU, then the algorithm looks at the $Cost(A_i, S_j, P_k)$ for all k, sorts the CPUs by increasing order of that cost and proceeds with the proposals. The profits for the proposals are computed from these costs and the knapsacks are based on those profits. Thus, the algorithm is not tied to a particular cost function. This allows the user or administrator to capture special affinities between application, storage, CPU triples by controlling the cost function.

3. Application prioritization and policy constrained placement: In a real data center, all applications are not treated equally. Some high-priority applications require the best storage, the fastest I/O paths and the most powerful servers.

Administrators also have other policies that dictate if certain applications can or cannot be placed on certain types of nodes. These constraints can be automatically encapsulated in the cost function by giving highly preferred triples a low cost value and conversely incompatible application, storage, CPU triples a high cost value. Incorporating such constraints in other algorithms like INDV-GR, PAIR-GR is complex as their greedy approaches are not flexible enough to account for such cost metrics.

4. Ability to improve existing configurations, accounting for migration costs: Another important characteristic of SPARK is its ability to start from any existing configuration and account for migration costs in making placement decisions. This helps design placement plans that do not require extensive data and VM migration. This is a very desirable feature and is in contrast to other planning approaches[4] that make decisions oblivious to the current configurations. One way to accommodate these in SPARK is by augmenting the $Cost$ function as follows. For each application A_i with storage at S_j and CPU at P_k, one can add a factor ($\psi * A_i.StgReq * C_{S_j \rightarrow S_{j'}}$) to every $Cost(A_i, S_{j'}, P_*)$ value where $0 \leq \psi \leq 1$ is a customizable parameter that modulates the bias in deciding to migrate or not (ψ=0 implies no bias and each application is free to move) and $C_{S_j \rightarrow S_{j'}}$ is the cost of migrating storage from S_j to $S_{j'}$. Similarly for job migration costs from P_k. We envision ψ to be decided based on the desired speed of orchestrating the final SPARK plan. For example, to handle a workload surge in a rapid manner, a high ψ value would prevent extensive VM and data migration.

5. Handling dynamic scenarios: SPARK is quick in generating placement decisions (see §4). This makes it highly suitable for use in response to dynamic SAN data center scenarios. Below we briefly outline how one might use it in response to workload surges, planned downtimes, growth and node failures.

- **Surges:** Once a workload surge is identified using the VM and SAN management tools (§2), SPARK can be initiated to quickly plan application storage and CPU placement with the new amplified workload. The ψ can be set to a high value to limit the amount of VM/data migration suggested by SPARK. Once SPARK outputs a placement plan, the orchestrator can automatically execute necessary migrations stabilizing the system.
- **Planned Downtimes:** If planning sufficiently in advance for scheduled downtimes of hardware (for maintenance or upgrades), it might be preferable to have a more optimized solution than to limit migration. For such planning, ψ is set low and the SPARK can be set to plan only with those SAN resources that will be up during that time. This output plan can then be executed and the system adequately prepared for the node downtimes.
- **Planned Growth:** Similar to downtimes, growth in application storage capacities or CPU requirements can be planned adequately in advance.
- **Node Failures:** SPARK can also similarly handle node failures as long as information about node state (e.g. for a CPU resource node, the state of the VM) can be recreated. For stateless applications and when technologies exist to recreate state, SPARK can output a plan fast and minimize migration by setting a high ψ value.

6. Based on well-studied problems: The fact that SPARK is fundamentally grounded in well-studied problems gives us hope that it will work well in a variety of situations and offer an opportunity for a more concrete theoretical analysis in future. The experimental section validates the belief of superior performance of SPARK in optimization quality as well as running time.

[4]For example, for an LP formulation of this problem, the increased number of constraints would make the tractable problem size even smaller.

5.2 Directions for Future Work

We believe SPARK to be a promising step towards addressing many data center management problems. With its versatility and quick, high-quality allocation plans, SPARK can assist administrators in developing a dynamic SAN infrastructure that meets performance goals with minimal manual supervision. However, there remain some interesting avenues of research towards achieving this utility-computing goal:

- **Resource Allocation Dimensionality**: In our current design, resources are allocated to a node based on a single resource dimension – either CPU for processing nodes or storage for storage nodes. However, there are situations when a higher dimensionality of allocation is desired. For example, a 2-D allocation to storage nodes based on its storage capacity and also the I/O throughput capacity. Such an allocation would require a two dimensional knapsack algorithm at each storage node, allocating applications ensuring that neither of the node capacities are violated. Higher dimensionality knapsacks are known to be inefficient, and SPARK might need new approximation techniques to maintain its high efficiency.
- **Network Modeling**: SPARK currently does not account for the underlying fabric design in the storage area network. Appropriately incorporating network models and flows into the framework would be a key improvement and is an interesting direction of future research.
- **Designing Cost Models**: One of the salient feature of SPARK is its flexibility to work with various cost models. In this paper, we described an example cost model that aims at improving application performance by allocating applications with higher I/O requirements closer to storage. Developing other relevant cost models for real data center environments is a promising direction with immediate impact.
- **Improved Profit Functions**: In §3.4 we described a technique for evaluating the *profit* for each proposal. That function can be designed specifically to meet certain objectives of allocation. For example, to aggressively seek better performance of the algorithm or to enhance chances of finding some feasible solution. A deeper understanding of the design of such functions and their impact on SPARK is also an interesting area for future work.

6 Conclusions

In this paper we presented a novel algorithm, called SPARK that optimizes coupled placement of application computational and storage resources on nodes in a modern virtualized SAN environment. By effectively handling proximity and affinity relationships of CPU and storage nodes, SPARK produces placements that are within 4% of the optimal lower bounds obtained by LP formulations. Its fast running time even for very large instances of the problem make it especially suitable for frameworks that autonomically deal with dynamic scenarios like workloads surges, scheduled downtime, growth and node failures. Among its other features, SPARK can iteratively improve any existing resource placement, accounting for migration costs and has inherent built-in flexibility to handle various placement constraints that impact the choice of resources where an application can be placed.

SPARK's good optimization quality and grounding in two well-studied theoretical problems seems to indicate a potential deeper theoretical connection and is an interesting area for analysis. Other interesting area is to consider load balancing and scheduling of computational resources along with placement.

References

[1] Altiris Inc. http://www.altiris.com/.

[2] Aurema: Application performance and server consolidation solutions. http://www.aurema.com/.

[3] CPLEX 8.0 student edition for AMPL.

[4] EMC Control Center Family. http://www.emc.com/products/storage_management/controlcenter.jsp.

[5] IBM System Storage DS8000. http://www-03.ibm.com/servers/storage/disk/ds8000/.

[6] IBM TotalStorage Productivity Center. http://www-306.ibm.com/software/tivoli/products/totalstorage-data/.

[7] IDC Quarterly Study, March 2006. http://www.bsa.org/idcstudy/.

[8] Knapsack problem: Wikipedia. http://en.Wikipedia.org/wiki/Knapsack_problem.

[9] Minknap Source Code. http://www.diku.dk/~pisinger/codes.html.

[10] Multi-commodity Flow: Wikipedia. http://en.Wikipedia.org/wiki/Multi-commodity_flow_problem.

[11] NEOS server for optimization. http://www-neos.mcs.anl.gov.

[12] NEOS server: MINOS solver. http://neos.mcs.anl.gov/neos/solvers/nco:MINOS/AMPL.html.

[13] Open Virtualization Standards. http://www.vmware.com/news/releases/community_source.html.

[14] Platespin: Server consolidation and disaster recovery with virtual machines. http://www.platespin.com/.

[15] Stable marriage problem: Wikipedia. http://en.Wikipedia.org/wiki/Stable_marriage_problem.

[16] Tiburon. http://www.almaden.ibm.com/StorageSystems/Advanced_Storage_Systems/Tiburon.

[17] VMware Inc. http://www.vmware.com/.

[18] VMware Server. www.vmware.com/products/server/.

[19] XenEnterprise: Virtual data center. http://www.xensource.com/solutions/virtual_data_center.html.

[20] Cisco MDS 9000 Family SANTap Service - Enabling Intelligent Fabric Applications. CISCO Whitepaper, 2005.

[21] VMware Infrastructure 3 architecture. VMware Whitepaper, 2006.

[22] VMware Infrastructure: Resource management with VMware DRS. VMware Whitepaper, 2006.

[23] B. Adra, A. Blank, M. Gieparda, J. Haust, O. Stadler, and D. Szerdi. Advanced POWER Virtualization on IBM eserver P5 Servers: Introduction and Basic Configuration. IBM Redbook, October 2004.

[24] G.A. Alvarez, J. Wilkes, E. Borowsky, S. Go, T.H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, and A. Veitch. Minerva: An Automated Resource Provisioning Tool for Large-Scale Storage Systems. *ACM Transactions on Computer Systems*, 19(4):483–518, November 2001.

[25] E. Anderson, J. Hall, J. Hartline, M. Hobbs, A.R. Karlin, J. Saia, R. Swaminathan, and J. Wilkes. An Experimental Study of Data Migration Algorithms. In *Proceedings of International Workshop on Algorithm Engineering*, pages 28–31, 2001.

[26] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running Circles Around Storage Administration. In *Proceedings of USENIX Conference on File and Storage Technologies*, pages 175–188, 2002.

[27] E. Anderson, M. Kallahalla, S. Spence, R. Swaminathan, and Q. Wang. Ergastulum: quickly finding near-optimal storage system designs. *HP Laboratories SSP technical report HPL-SSP-2001-05*, June 2002.

[28] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, DP Pazel, J. Pershing, and B. Rochwerger. Oceano - SLA-based Management of a Computing Utility. In *Proceedings of IFIP/IEEE Symposium on Integrated Network Management*, pages 855–868, 2001.

[29] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[30] S. Ceri, G. Martella, and G. Pelagatti. Optimal File Allocation in a Computer Network: A Solution Method Based on the Knapsack

Problem. *Computer Networks*, 6(5):345–357, November 1982.

[31] J.S. Chase, D.C. Anderson, P.N. Thakar, A.M. Vahdat, and R.P. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of ACM Symposium on Operating System Principles*, pages 103–116, 2001.

[32] C. Chekuri and S. Khanna. A PTAS for the multiple knapsack problem. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, pages 213–222, 2000.

[33] C. Chekuri, S. Khanna, and F.B. Shepherd. The all-or-nothing multicommodity flow problem. In *Proceedings of ACM Symposium on Theory of Computing*, pages 156–165, 2004.

[34] C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*, 2005.

[35] L. W. Dowdy and D. V. Foster. Comparative Models of the File Assignment Problem. *ACM Surveys*, 14:287–313, 1982.

[36] Ian Foster. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. In *Proceedings of CCGRID*, 2001.

[37] Ian Foster. An Open Grid Services Architecture for Distributed Systems Integration. In *Proceedings of ICPP*, 2002.

[38] J. Hall, J. Hartline, A.R. Karlin, J. Saia, and J. Wilkes. On Algorithms for Efficient Data Migration. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, pages 620–629, 2001.

[39] L. Huston, R. Sukhtankar, R. Wickremesinghe, M. Satyanarayanan, G. Ganger, E. Riedel, and A. Ailamaki. Diamond: A Storage Architecture for Early Discard in Interactive Search. In *Proceedings of USENIX Conference on File and Storage Technologies*, pages 73–86, 2004.

[40] O.H. Ibarra and C.E. Kim. Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems. *Journal of the ACM* , 22(4):463–468, 1975.

[41] S. Khuller, Y.A. Kim, and Y.C.J. Wan. Algorithms for Data Migration with Cloning. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 27–36, 2003.

[42] P. Klein, A. Agrawal, R. Ravi, and S. Rao. Approximation through multicommodity flow. In *Proceedings of Symposium on Foundations of Computer Science*, pages 726–737, 1990.

[43] C. Lu, G.A. Alvarez, and J. Wilkes. Aqueduct: Online Data Migration with Performance Guarantees. In *Proceedings of USENIX Conference on File and Storage Technologies*, 2002.

[44] D. G. McVitie and L. B. Wilson. The Stable Marriage Problem. *Communications of the ACM*, 14(7):486–490, July 1971.

[45] J. Mears. LinuxWorld Virtualization goes beyond server consolidataion. *Network World*, August 2006.

[46] J. Mears. VMware beyond the basics. *Network World*, August 2006.

[47] M. Nelson, B. Lim, and G. Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of USENIX Annual Technical Conference*, pages 383–386, 2005.

[48] D. Pisinger. A Minimal Algorithm for the 0-1 Knapsack Problem. *Journal of Operations Research*, 45:758–767, 1997.

[49] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of IEEE Symposium on High Performance Distributed Computing*, 1998.

[50] R. Raman, M. Livny, and M. Solomon. Policy driven heterogeneous resource co-allocation with Gangmatching. In *Proceeding of IEEE Symposium on High Performance Distributed Computing*, pages 80–89, 2003.

[51] A. Romosan, D. Rotem, A. Shoshani, and D. Wright. Co-Scheduling of Computation and Data on Computer Clusters. In *Proceedings of SSDBM*, 2005.

[52] M. W. P. Savelsbergh. A Branch-and-Price Algorithm for the Generalized Assignment Problem. *Journal of Operations Research*, 45(6):831–841, 1997.

[53] D.B. Shmoys and É. Tardos. An Approximation Algorithm for the Generalized Assignment Problem. *Journal of Mathematical Programming*, 62(1):461–474, 1993.