Coupled Placement in Modern Data Centers

Madhukar Korupolu
IBM Almaden Research Center
madhukar@us.ibm.com

Aameek Singh
IBM Almaden Research Center
aameek.singh@us.ibm.com

Bhuvan Bamba
Georgia Tech
bhuvan@cc.gatech.edu

Abstract

We introduce the coupled placement problem for modern data centers spanning placement of application computation and data among available server and storage resources. While the two have traditionally been addressed independently in data centers, two modern trends make it beneficial to consider them together in a coupled manner: (a) rise in virtualization technologies, which enable applications packaged as VMs to be run on any server in the data center with spare compute resources, and (b) rise in multi-purpose hardware devices in the data center which provide compute resources of varying capabilities at different proximities from the storage nodes.

We present a novel framework called CPA *for addressing such coupled placement of application data and computation in modern data centers. Based on two well-studied problems – Stable Marriage and Knapsacks – the* CPA *framework is simple, fast, versatile and automatically enables high throughput applications to be placed on nearby server and storage node pairs. While a theoretical proof of* CPA*'s worst-case approximation guarantee remains an open question, we use extensive experimental analysis to evaluate* CPA *on large synthetic data centers comparing it to Linear Programming based methods and other traditional methods. Experiments show that* CPA *is consistently and surprisingly within* 0 *to* 4% *of the Linear Programming based optimal values for various data center topologies and workload patterns. At the same time it is one to two orders of magnitude faster than the LP based methods and is able to scale to much larger problem sizes.*

The fast running time of CPA *makes it highly suitable for large data center environments where hundreds to thousands of server and storage nodes are common. LP based approaches are prohibitively slow in such environments.* CPA *is also suitable for fast interactive analysis during consolidation of such environments from physical to virtual resources.*

1. Introduction

Managing enterprise data centers is a challenging and expensive task [9]. Modern data centers often have hundreds to thousands of servers and storage subsystems, connected via a web of network switches. As the scale and complexity continues to increase, enterprises find themselves at a unique tipping point. The recent success of virtualization technologies is encouraging enterprises to transform their data centers from the old rigid IT configurations to a more *agile* infrastructure using virtualization technologies. Server virtualization technologies like VMware and Xen enable applications, which were hitherto running in their individual silos of servers (that were often over-provisioned), to be packaged as virtual machines (VMs) that can run on any compatible physical machine in the data center. It also allows multiple virtual machines to be run on the same physical machine without interference as well as moving a VM from one machine to another without application downtime [29], [22]. Such agility while improving utilizations and reducing costs, introduces new challenges.

Imagine an enterprise migrating its applications from a physical IT infrastructure to a virtualized *cloud* within the enterprise. One of the tasks that the administrators need to plan intelligently during this consolidation is to decide where to place application computation (VM) as well as application data, among the available server and storage resources in the virtualized data center. The traditional approach of making computation and data placement decisions independently [34], [14], [16] yields suboptimal placements due to different proximities of server and storage nodes in the data center. Often times data center administrators work around this problem by over-provisioning resources which leads to under-utilization, not only of capital and hardware resources but also of space and energy costs. As a result, the need for an intelligent placement planning mechanism is significant.

Another data center reality complicates this process further. Data centers evolve over time and not all nodes in the data center are created equal. Figure 1 shows an example modern data center with servers, storage and network nodes. Different nodes could be purchased at different times thus having differing capabilities based on the technology at that time. For example in Figure 1, one switch can be more recent than the other and have lower latency and greater I/O throughput capacity. This leads to differing *affinities* among different server-storage node pairs. This heterogeneous nature of data center environments requires handling storage and computational resources in a coupled manner.

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

For example, placing an application computation (VM) on a server node needs to take into account the affinity of the node to application's storage node. A practical placement planning technique must be able to take these into account.

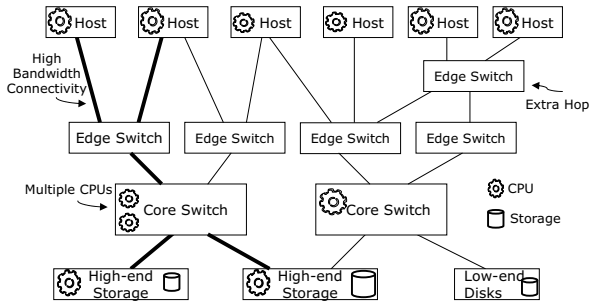

Figure 1. Modern data center with heterogeneous resources

Further, an interesting trend of late is that of multi-purpose hardware devices. For example some storage devices (e.g., IBM DS8000 series) now come with a built-in compute resource (POWER5 logical partitioning LPAR) which can be used for hosting applications [7], [12]. Similarly many network switch devices (e.g., Cisco MDS 9000 switches [8]) have additional x86 processor blades that can be fitted with modules to enable functionality offload to these switches.

The presence of such multi-purpose devices provides additional chances for optimization, as applications that require higher throughput of data can benefit by running closer to storage. However taking advantage of this proximity requires careful planning mechanisms that place application data and computation in an integrated manner so that both can go on nearby node pairs.

1.1. The Coupled Placement Problem

To capture the above discussion we now introduce the coupled placement problem.

Let $\mathcal{A} = \{A_i : 1 \leq i \leq |\mathcal{A}|\}$ denote the set of applications that need to run in the data center. Each application requires a certain amount of compute resource (denoted by $A_i.Creq$, in CPU cycles per second), certain amount of storage resource (denoted by $A_i.Sreq$, in Gigabytes) and certain rate of data transfer from storage to the compute resource (denoted by $A_i.Iorate$, in post-cache IOs per second or Mbps). The computation part packaged as a virtual machine has to be placed on a compute node and the storage part has to be placed on a storage node in the data center.

Let the set of compute nodes in the data center be denoted by $\mathcal{C} = \{C_j : 1 \leq j \leq |\mathcal{C}|\}$. This includes all the available compute resources in the data center where an application can be run. This could be a host server node, a switch node with available processing power or a storage node with available processing power. Each such compute resource C_j has an associated upper limit, denoted by $cap(C_j)$, capturing the amount of processing power capacity it has, in processor cycles per second. Similarly let the set of storage resource nodes in the data center be denoted by $\mathcal{S} = \{S_k : 1 \leq k \leq |\mathcal{S}|\}$. Each such storage resource S_k has an associated upper limit, denoted $cap(S_k)$, capturing the amount of storage space capacity it has, in Gigabytes.

These compute and storage resources are connected to each other directly or indirectly by a high speed network as shown in example Figure 1. Let $dist(C_j, S_k)$ denote the *distance* between the nodes C_j and S_k, either in terms of number of hops, or latency or any reasonable measure. Let $\boldsymbol{Cost(A_i, C_j, S_k)}$ denote the cost incurred for an application A_i if its computation were placed on node C_j and storage on node S_k. This can be obtained either as measurement interpolation from the real system or as user input. A sample cost function for example is $A_i.Iorate * dist(C_j, S_k)$, favoring high I/O rate applications to be on nearby node pairs. More generally, the cost function $\boldsymbol{Cost : \mathcal{A} \times \mathcal{C} \times \mathcal{S} \rightarrow \mathbb{R}}$ can be any user given function. More about the cost function in §3.

Given this, the goal of the coupled placement problem is to place the computation and data of each application among the nodes of the data center such that total cost over all applications is minimized, i.e.,

$$min \sum_i Cost(A_i, Cnode(A_i), Snode(A_i)) \qquad (I)$$

subject to not violating the capacity constraints at any of the storage and compute nodes:

$$\forall j \sum_{i:Cnode(A_i)=C_j} A_i.Creq \leq cap(C_j)$$

$$\forall k \sum_{i:Snode(A_i)=S_k} A_i.Sreq \leq cap(S_k)$$

where $Cnode(A_i)$ denotes the node where the computation for A_i is placed and $Snode(A_i)$ denotes the node where the storage for A_i is placed[1].

1.2. State of the Art and Limitations

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

Ergastulum [16], Hippodrome [15] and Minerva [14] attempt to place storage for applications assuming the application computation locations are fixed. Oceano [17] and Muse [19] provide methods for provisioning computation resources in a data center but they do not have a mechanism to take into account the underlying storage system and

1. For the sake of notational consistency, throughout this paper, we use i as an index over applications (so $1 \leq i \leq |\mathcal{A}|$), j as an index over compute nodes ($1 \leq j \leq |\mathcal{C}|$), and k as an index over storage nodes ($1 \leq k \leq |\mathcal{S}|$).

its affinities to compute nodes. Similarly, File Allocation problems [23] look at where to allocate files/storage assuming computation locations are already fixed. Generalized Assignment problems [34] look at assigning tasks to processors (compute nodes) but these again are for single resource allocation and not coupled storage and computation allocation.

Recent products like VMware Infrastructure [10] and its Dynamic Resource Scheduler (DRS) [11] address monitoring and allocation of only computation resources in a data center. Similarly research efforts like [37], [38] address server consolidation and dynamic VM placement. While these are good starting points, they only concentrate on allocation of one resource while ignoring the other. Such single-resource allocation approaches cannot take advantage of the multi-purpose resources and proximities in modern virtualization-enabled data centers.

One promising approach to model the simultaneous placement of computation and storage may appear to be to use the min-cost version of network flows [13]. However this would require multi-commodity flow version of network flows as each application would have to be its own commodity – in order to keep its requirements separate from those of other applications. Multi-commodity flow problems however are known to be hard to solve in practice even for medium sized instances [27]. Furthermore, they would end up splitting each application's computation and storage requirements among multiple resource nodes which may not always be desirable. If such splitting is to be avoided, then we would need unsplittable min-cost multi-commodity flows [21], which are even harder in practice.

Another area of related work is that of co-scheduling data and computation in grid computing [31], [32], [33]. The aim of grid computing in general has been to pool together resources from disparate geographically distributed IT environments over wide area networks [25], [24]. Matchmaking and Gangmatching [31], [32] provide protocols in such environments for jobs to advertise their needs and for resources to advertise their capabilities and policies and then they provide mechanisms for jobs to approach resources for a potential match. The focus is more on schemes to decide policy match when there is such decentralized ownership and not on optimization. Such decentralized ownership is not an issue in data centers. [33] builds on [31] 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 nearness optimization when placement on the same node is not possible. The grid approaches also employ replication strategies often to create replica of the data closer to the compute node. Such aggressive replication, though justifiable in wide-area grid computing environments is not practical in data center environments.

1.3. Our Contributions

In this paper, we present a novel framework called CPA (Coupled Placement Advisor) for addressing the coupled placement of application storage and computation in modern data centers.

The problem of placing computation and storage simultaneously involves multiple interesting aspects. For example, if multiple applications compete for the same compute or storage node, then which ones should the node select for its best utilization? This is akin to the Knapsack problem [3]. Similarly, applications have to decide which nodes to compete for – whether to compete for nodes that are easier to get or for nodes that they prefer the most. Here we borrow ideas from the Stable Marriage problem [6] to choose the most "stable" matches.

We evaluate CPA experimentally, comparing it to Linear Programming based optimal solutions and other candidate algorithms both in terms of solution quality and running time for a wide range of workloads and topologies. We find that CPA is consistently **within** 0 **to** 4% **of the optimal LP based values**. The LP based approaches tend to be slow and could run to completion only for small input sizes. CPA, however, is faster by one to two orders of magnitude and is able to handle much larger problem sizes. Comparison to other candidate algorithms shows that CPA is about an order of magnitude faster than the next best candidate algorithm for large instances. It also yields placements that are better by 30 to 40%. The fast running time of CPA enables it to scale well for modern data centers which regularly have hundreds to thousands of compute nodes, storage nodes and applications.

The other salient feature of CPA is its **versatility**. This enables it to accommodate various system policies and application preferences that often arise in practical deployments: accounting for already existing placements, accounting for migration costs and improving existing configurations iteratively, dealing with dynamic data center complexities etc. Given the constantly changing nature of real data centers, such flexibility is often a necessity in practice.

Next, we start with a few preliminaries for the coupled placement problem.

2. Coupled Placement Preliminaries

The coupled placement problem was defined in Section 1.1. Given a set $\mathcal{A}$ of applications, a set $\mathcal{C}$ of compute nodes and set $\mathcal{S}$ of storage nodes in the data center, and a cost function $\boldsymbol{Cost : \mathcal{A} \times \mathcal{C} \times \mathcal{S} \rightarrow \mathbb{R}}$, the coupled placement problem is to place the computation and data of each application among the nodes in the data center so as to minimize the total cost over all applications, i.e.,

$$min \sum_i Cost(A_i, Cnode(A_i), Snode(A_i)) \qquad (I)$$

subject to not violating the capacity constraints at any of the storage and compute nodes:

$$\forall j \sum_{i:Snode(A_i)=S_k} A_i.Sreq \leq cap(S_k)$$

$$\forall k \sum_{i:Cnode(A_i)=C_j} A_i.Creq \leq cap(C_j)$$

We begin with a brief discussion of the cost function before proving that the coupled placement problem is NP-Hard.

Cost Function: The idea behind the cost function is that it captures the cost incurred for each application when its computation is placed on a certain compute node and storage is placed on a certain storage node. This can be any function that suitably captures the affinities that exist between applications and various storage, compute nodes (for example, based on latency between server and storage node, dollar cost of storage and processors, application priorities, node types etc). For the sake of concreteness, we use the following example cost function in this paper[2].

$$Cost(A_i, C_j, S_k) = A_i.Iorate * dist(C_j, S_k)$$

where $dist(C_j, S_k)$ is the distance between nodes C_j and S_k either in terms of number of hops, or latency or any reasonable measure. This function tends to keep high I/O rate applications on nearby compute-storage node pairs. For example, if a multi-purpose storage device (with compute and storage facilities on the same node) is available, then all else being equal, a high I/O rate application will incur a lower cost if both its computation and storage are placed on that device.

Theorem 2.1. *The Coupled Placement Problem is NP-Hard.*

Proof: We reduce the Knapsack Problem [3] (§3.1) to the coupled placement problem. The idea is that even if a simpler case of the coupled placement problem involving only one storage node and two compute nodes can be solved, then it can be used to solve the knapsack problem. Given an instance KI of the knapsack problem – a collection of n items $a_1, \ldots a_n$, with item a_i having a weight w_i and value v_i, and a sack of capacity W – we construct an instance CPPI of the coupled placement problem as follows:

- Setup n applications, $\mathcal{A} = \{A_1, \ldots, A_n\}$, one corresponding to each item.
  - For application A_i, set $A_i.Creq = w_i$, $A_i.Sreq = 1$, and $A_i.Iorate = v_i$
- Setup one storage node, $\mathcal{S} = \{S_1\}$ with $cap(S_1) = n$.
- Setup two compute nodes, $\mathcal{C} = \{C_1, C_2\}$ with $cap(C_1) = W$ and $cap(C_2) = \infty$.
- Set $dist(C_1, S_1) = 1$ and $dist(C_2, S_1) = 2$.

It is easy to see that the contents of the knapsack correspond to the computations placed on C_1 and vice-versa, and that a solution to the CPPI yields a solution to KI. Thus an algorithm for solving the coupled placement problem can be used to solve the knapsack problem. Hence it follows that the coupled placement problem is NP-Hard. □

The proof outline above shows that even a simpler case of only two compute nodes with fixed storage makes the problem NP-Hard. Having to decide coupled placements for both storage and computation with general cost, distance functions makes the problem more complex. We highlight a few interesting special cases of the problem before presenting the algorithms in the next section.

Special case of one resource fixed: If either compute 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 [23] for placing files/storage assuming compute location is fixed; (b) Minerva, Hippodrome [14], [15] planning SAN storage design assuming compute locations are fixed; and (c) Generalized Assignment problems [34]: Assigning tasks to processors ignoring storage altogether.

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 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 (C_j, S_k) pairs then the problem could be simplified to placing storage and computation 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 Section 4.

3. Algorithms

In this section, we begin by outlining two simpler algorithms – a greedy individual placement algorithm INDV-GR that places computation and storage of applications independently in a natural greedy fashion and a pairs placement algorithm PAIR-GR that greedily places each application's computation and 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 $\frac{App.Iorate}{App.Sreq}$ and greedily assigning them to storage nodes sorted by $BestDist$ metric, where

2. It should be noted however that the CPA algorithm and the framework described in §3 are flexible to work with any user defined cost function. For more discussion about this versatility, please see §4.7.

$BestDist(S_k) = min_j dist(C_j, S_k)$. Intuitively, INDV-GR tries to place highest throughput applications (normalized by their storage requirement) on storage nodes that have the closest compute nodes. In the next phase, it will similarly place application computation on the compute nodes.

Algorithm 1 INDV-GR: Greedy Individual Placement

```
1: RankedAppsStgQ ← Apps sorted by App.Iorate / App.Sreq // decreasing.
2: RankedStgQ ← Storage nodes sorted by BestDist metric // increasing.
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.Sreq ≤ 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 Computation placement
```

However, due to its greedy nature, a poor placement of applications can result. For example, say there is a highly preferred storage node S_k with capacity 800 units. And there are three applications A_1, A_2, A_3 with storage requirements of 600, 500, and 300 units respectively and Iorates of 1200, 900 and 500 units respectively. Then the greedy strategy causes it to select A_1 whereas the $A_2 + A_3$ combination is better. The INDV-GR also does not account for storage-compute node affinities beyond using a rough $BestDist$ metric. For example, if A_i storage is placed on S_k, INDV-GR does not especially try to place A_i computation on the node closest to S_k.

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 $\frac{App.Iorate}{App.Creq * App.Sreq}$ on storage, compute node **pairs** sorted by the distance between the nodes of the pair. With this, applications are placed simultaneously into storage and compute node buckets based on their affinity as 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 a poor placement (e.g., selecting a 600 but losing out on a 500 plus 300 combination). Ideally, each storage (and compute) node should be able to select application *combinations* that best minimize the overall cost value of the system. This hints at usage of **Knapsack**-like algorithms [3]. 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 orders indicates a connection to the **Stable-Marriage** problem [6] described below.

Algorithm 2 PAIR-GR: Greedy Pairs Placement

```
1: RankedAppsQ ← Apps sorted by App.Iorate / App.Creq*App.Sreq // decreasing
2: RankedPairsQ ← {C x S} sorted by distance // increasing
3: while RankedAppsQ ≠ Ø do
4:    App ← RankedAppsQ.pop()
5:    for (i=0; i<RankedPairsQ.size; i++) do
6:       S_k ← RankedPairsQ[i].Snode()
7:       C_j ← RankedPairsQ[i].Cnode()
8:       if (App.Sreq ≤ S_k.AvlSpace AND App.Creq ≤ C_j.AvlCPU) then
9:          Place App storage on S_k, App computation on C_j
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
```

3.1. The CPA 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 CPA. So we begin by a brief description of these problems.

Knapsack Problem[3], [30]: Given n items, a_1 through a_n, each item a_i has weight w_i and a profit value v_i. The total capacity of the knapsack is W. The 0-1 knapsack problem asks for the collection of items to place in the knapsack so as to maximize the profit. Mathematically:

$$max \sum_{i=1}^{n} v_i x_i \;\; subject\ to \;\; \sum_{i=1}^{n} w_i x_i \leq W$$

where x_i = 0 or 1 indicating whether item a_i is selected or not. This problem is known to be NP-Hard and has been well-studied for heuristics of near optimal practical solutions [26].

Stable Marriage Problem [6], [28]: Given n men and n women, where each person has a ranked preference list of the members of the opposite group, the stable marriage problem seeks to pair the men and women in such a way that there are no two people of opposite group who would both rather have each other than their pair partners. If there are no two 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/her ranked list of preferred medical universities and each university submits its ranked list of preferred applicants.

The Gale-Shapely *Proposal algorithm* [28] is the one that is commonly used in such problems. It involves a number 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 on her preference list than her current best and rejects otherwise. The man who is rejected becomes unengaged and moves to the next in his preference list. This iterative process is known to yield provably stable results [28].

CPA Approach: Based on the above two, we now describe the CPA approach. Consider a general scenario where the compute 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 C_j be the compute node where the computation of A_i is currently placed. Then all S_k, $1 \leq k \leq |\mathcal{S}|$ are ranked in increasing order of $Cost(A_i, C_j, S_k)$[3].

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). After each application has made its proposal in this manner, on the receiving end, each storage node looks at all the proposals it has received. For each such proposal, it computes a profit value measuring the utility of that proposal. How to compute these profit values is discussed in §3.2. Now among all the storage nodes, we pick the node that received the highest cumulative profit value in proposals and do a knapsack computation for that node[4]. This knapsack computation chooses from the set of proposals received at the node – each of which now has a profit value and a size requirement $A_i.Sreq$ – a subset of proposals to maximize the value while staying below the node's capacity limit. The chosen proposals are considered accepted at this node. The other application proposals to the node are considered rejected. The rejected ones move down their preference list and propose to the next candidate. This process repeats until all applications are accepted. This repeated proposal-knapsack scheme for storage nodes constitutes one CPA-Stg phase. The pseudocode for this part is given in Algorithm-3.

We assume a dummy storage node S_{dummy} (similarly dummy compute node C_{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.

3. Or in case computations have not been placed (for example, when doing a fresh placement) we use the $BestDist(S_k)$ metric.

4. Though the Knapsack problem is NP-Hard, there are known polynomial time approximation schemes (PTAS) for it [20] which work reasonably well in practice to give not exactly optimal but close to optimal solutions. We use one such package [1] here.

Algorithm 3 CPA-Stg: Storage placement in CPA

```
1: for all App in AppsQ do
2:    Create preference list of storage nodes for App sorted by
      Cost(App, Cnode(App), *) // incr.
3:    Propose to best storage node in the list
4: end for
5: while (All apps not placed) do
6:    MaxStgNode ← StgQ[0]
7:    for all Stg in StgQ do
8:       Compute profits for proposals received at Stg
9:       MaxStgNode ← argmax(MaxStgNode.profit ,
         Stg.profit)
10:   end for
11:   Knapsack MaxStgNode
12:   for all Apps accepted by Knapsack MaxStgNode do
13:      Place App storage on MaxStgNode
14:   end for
15:   for all Rejected apps do
16:      Propose to next storage node in its preference list
17:   end for
18: end while
```

CPA-Compute Phase: This phase is similar to the CPA-Stg phase above but with compute nodes substituted for storage nodes. Given the current storage placements, the CPA-Compute phase decides the computation placements for applications based on the affinities from the chosen storage locations. The pseudocode for this part is similar to the one in Algorithm-3.

Example: Figure 2 gives an illustration for the working of the CPA-Stg and CPA-Compute phases in rounds. CPA-Stg brings A_2-storage closer to A_2-computation and CPA-Compute further improves by bringing A_2-computation closer to A_2.Stg. Knapsacks help choose the $A_1 + A_2$ combination over A_3 during placement.

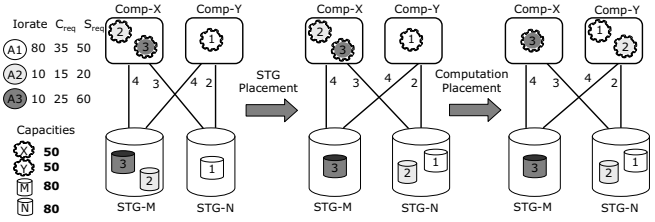

Figure 2. Placement in rounds. STG placement brings A_2-storage closer to A_2-computation and Compute placement further improves by bringing A_2-computation closer to A_2-storage. Knapsacks help choose A_1+A_2 combination over A_3 during placement.

CPA-Swap Phase: Though the combination of CPA-Stg and CPA-Compute phases addresses many possibilities well, there are certain scenarios like the one in Figure- 3 where they are limited. Here a move of either computation

or storage alone does not improve the placement but moving of both simultaneously does. This is where the CPA-Swap step comes in. It looks at pairs of applications A_i and $A_{i'}$ for which swapping their compute-storage node pairs improves the cost while staying below capacity limits and performs the swap. (e.g., Figure- 3).

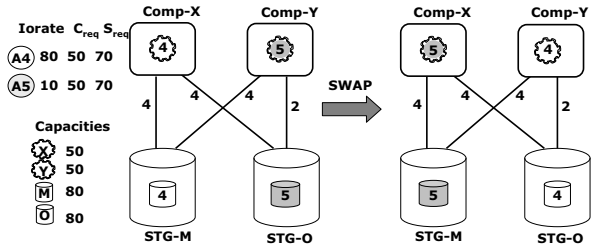

Figure 3. Swap exchanges storage-compute pairs between A_4, A_5. Individual rounds cannot make this move - during STG placement, M and O are equally preferable for A_1 as they have the same distance from A_4-computation (X). Similarly during Compute placement.

CPA Overall: Combining these insights, the CPA 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 compute, followed by a swap phase. It improves the solution iteratively, until a chosen termination criterion or a local optimum is reached.

Algorithm 4 CPA: Proposals-and-Knapsacks

```
MinCost ← ∞, SolutionCfg ← Ø
loop
    CPA-Stg()
    CPA-Comp()
    CPA-Swap()
    Cost ← 0
    for all App in AppsQ do
        Cost ← Cost + Cost(App, Cnode(App), Snode(App))
    end for
    if (Cost < MinCost) then
        MinCost ← Cost
        SolutionCfg ← current placement solution
    else
        break // termination by local optimum
    end if
end loop
return SolutionCfg
```

3.2. Computing Profit Values

One of the key steps in the CPA algorithm is how to compute the profit values for the proposals. Recall that when a storage node S_k (or a compute node C_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 has not found a location yet). If it does, say at node $S_{k'}$ ($S_{k'}$ must be below S_k in A_i's preference list, otherwise A_i would not have proposed to S_k.) Then the receiving node S_k would look at how much the system would save in cost if it were to accept A_i. This is essentially $Cost(A_i, C_j, S_{k'}) - Cost(A_i, C_j, S_k)$ where C_j is the current (fixed for this storage phase) location of A_i's computation. This is taken as the profit value for A_i's proposal to S_k.

On the other hand, if A_i does not have any storage location or if A_i has storage at S_k itself, then the receiving node S_k would like to see how much more the system would lose if it did not select A_i. If it knew which storage node $S_{k'}$, A_i would end up on if it was not selected by S_k, then the computation is obvious. Just taking a difference as above from $S_{k'}$ would give the necessary profit value. However where in its preference list A_i would end up if S_k rejects it, is not known at this time. In the absence of this knowledge, a conservative approach is to assume that if S_k 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 $Cost(A_i, C_j, S_{dummy}) - Cost(A_i, C_j, S_k)$.

An aggressive approach is to assume that A_i would get selected at the very next storage node in its preference list after S_k. In this approach, the profit value would then become $Cost(A_i, C_j, S_{k'}) - Cost(A_i, C_j, S_k)$ where $S_{k'}$ is the node immediately after S_k in the preference list for A_i. The reason this is aggressive is that $S_{k'}$ 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 CPA algorithm with this approach are very close (within 4%) to the optimal for a wide range of scenarios. In future, we plan to examine more sophisticated approaches including estimating probabilities that a given item would be accepted at a particular node based on history from past selections.

4. Evaluation

In this section, we evaluate the performance of the different algorithms through a series of simulation based experiments. Section §4.1 describes the algorithms considered, §4.2 describes the experimental setup followed by results in subsequent sections. We summarize our findings in §4.7.

4.1. Algorithms and Implementations

We compare the following algorithms in our evaluation.

- **Individual Greedy Placement (INDV-GR)**: The greedy algorithm that places application compute and storage independently – Algorithm-1 (ref. §3).

- **Pairwise Greedy Placement (PAIR-GR)**: The greedy algorithm that places applications into best available compute-storage node *pairs* – Algorithm-2 (ref. §3).
- **OPT-LP**: The optimal solution obtained using linear programming based methods. We use the CPLEX [2] package to solve the ILP, however it works only for the smallest problem size. Since ILPs are known to be hard in practice, we use the corresponding LP relaxation of the ILP for larger problem sizes. These yield fractional solutions which, though not always a valid solution for the original ILP, are nevertheless a good lower bound on the best that can be achieved for the ILP. It serves as a baseline to compare other algorithms with. We use the popular MINOS [5] solver (through NEOS [4] web service) with fractional solutions in the [0,1] range to solve the LP relaxations.
- **CPA**: The CPA algorithm as described in §3. It uses the 0/1 knapsack algorithm contained in [30] with source code from [1].
- **CPA-R1**: The CPA algorithm as described in §3 is an iterative one that runs until a local optimum is reached. CPA-R1 is a variant of CPA that runs only for one single round. This helps to understand the iteratively improving nature of CPA.

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 each experiment, the results are averaged over multiple runs.

4.2. Synthetic Data Center Generator

To evaluate the algorithms on large data centers, we designed a synthetic data center generator which takes as input the number of server nodes, the number of storage nodes and generates a industry standard SAN data center using the core-edge design pattern of connectivity. To avoid excessive parameterization, we use certain simple rules of thumb: For a given set of say N applications, we use a configurable fraction (here $N/3$) of server nodes, a configurable fraction (here $N/20$) of high-end storage nodes with built-in spare compute resources and a configurable fraction (here $N/25$) of regular storage devices without additional compute resources. The storage nodes are connected to server nodes through three levels of core and edge switches (similar to Figure 1) as in the best practice core-edge design. We use $N/9$ edge switches (one for every three servers), about $N/50$ mid-level core switches and a few (here $N/200$) high-end core switches with built-in spare compute resources.

Given an instance like this, the number of combinations for placing applications among compute and storage node pairs is $|\mathcal{A}| \times |\mathcal{C}| \times |\mathcal{S}|$. This is roughly the number of steps an algorithm like PAIR-GR would take to evaluate all combinations. We use this as a measure of the **problem complexity** or **problem size**.

The computation and storage requirements for applications are generated using a normal distribution with configurable mean and standard deviations. The mean and standard deviations are configured as an α or β fraction of the total capacity (compute or storage accordingly) in the system per application. More concretely, for compute resources, the mean (μ_c) equals $\frac{\alpha * N_c * cap(C)}{N}$ where α is a fraction between 0 and 1, N_c is the number of compute nodes in the system, $cap(C)$ is the capacity of each single compute node and N is the total number of applications in the system. The standard deviation (σ_c) equals $\frac{\beta * N_c * cap(C)}{N}$ where β is a fraction between 0 and 1. Similarly for storage nodes.

Experiments with increasing α and β are presented in §4.4 and §4.5. The varying α experiments investigate how CPA behaves as available resources are filled loosely or tightly by the application workloads. The varying β experiments investigate how CPA behaves with an increasing level of heterogeneity of the application workloads.

Another important piece of the input is the application compute-storage node distance matrix. For this, we use an exponential function based on the physical inter-hop distance; the closest compute-storage node pairs (both nodes inside a high end storage system) are set at distance 1 and for every subsequent level the distance value is multiplied by a *distance-factor*. A higher distance factor implies greater relative distance between two consecutive levels of compute nodes from the underlying storage nodes. Experiments with increasing distance factor are presented in §4.6.

4.3. Experiments with Increasing Data Center Size

An important requirement for CPA is to be able to scale to large data center environments. To study this, we evaluate the different algorithms with increasing sizes of the data center. Figure-4 shows the solution quality of the different algorithms as the problem size is increased up to 575M. At the highest point, this corresponds to about 2500 applications on about a 1000 servers, 125 of which are high-end storage nodes, 100 regular storage nodes, 280 edge switches and 12 core switches. The α and β parameters were set to 0.55 each and distance factor was set to 2.

The OPT-LP implementation could only solve up to a problem size of 1.08M ($N = 300$ applications) because of its inherent slowness as the number of variables and constraints becomes too many. We measure the running time and solution quality for each of the algorithms. Recall that the cost metric is given by the cumulative sum of the application iorate times its storage-compute pair distance.

Looking at Figure-4, depicts the solution quality of INDV-GR, PAIR-GR, CPA, and CPA-R1. This figure does not include OPT-LP since the latter only works for small problem sizes (upto 1.08M). We show them along with OPT-LP in a separate zoomed graph in Figure-5 focusing on problem

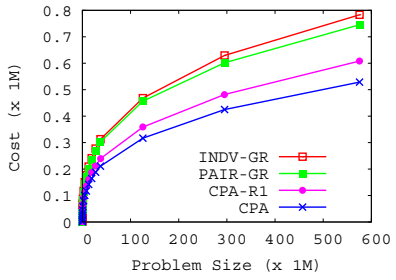

Figure 4. Quality with increasing size. *OPT-LP only works upto 1.1 M size. See Fig-5.*

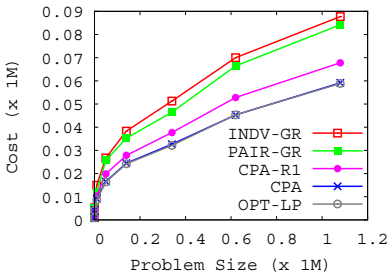

Figure 5. Comparison with OPT-LP. CPA *is within 0-4%. See Table-1*

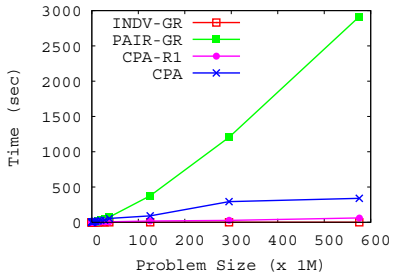

Figure 6. Time with increasing size. *Even for 1M size, OPT-LP took 30+ minutes.*

sizes upto 1.08M. The exact cost measures for these small sizes are given in Table-1.

First, from Figure-4 note the better performance of CPA and CPA-R1 compared to the two greedy algorithms. Of the two greedy algorithms, PAIR-GR does better as it places applications based on storage-computation pair distances, whereas INDV-GR's independent decisions fail to capture that. CPA-R1 performs better than both greedy algorithms. This is because of its use of knapsacks (enabling it to pick the right application combinations for each node) and placing computation based on the corresponding storage placement. Running CPA for further rounds improves the value iteratively to the best value shown.

Figure-5 shows the same experiment zooming in on the smaller problem sizes (up to 1.08M) where the OPT-LP was also able to complete. While the rest of the trends are similar, most interesting is the closeness in curves of OPT-LP and CPA, which are pretty much overlapping. Table-1 gives the exact values achieved by OPT-LP and CPA, CPA-R1 for these problem sizes. CPA is within 2% for all but one case where it is within 4% of OPT-LP. This is an encouraging confirmation of CPA's optimization quality.

Size	CPA-R1	CPA	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 only works upto 1.08 M due to solver [5] being unable to handle large number of variables and constraints.* CPA **is within 4% of OPT-LP and scales to much larger problem sizes.**

Running Time Comparisons: Figure-6 shows the time taken by the different algorithms. As expected, INDV-GR is extremely fast since it places application computation and storage independently, with a worst-case complexity of $O(|A| * (|S| + |C|))$. PAIR-GR, on the other hand, is the slowest of all. It generates all storage-computation pairs and places applications on these pairs which in the worst-case can be $O(|A| * |S| * |C|)$. The CPA algorithm on the other hand runs iteratively improving the storage and compute locations repeatedly using proposals and knapsacks until a local optimum is reached. The CPA-R1 curve shows the running time of CPA for just the first one round, which is very fast from Figure-6. The CPA algorithm itself runs for further rounds improving the solution even further and its total running time depends on the total number of rounds and the complexity of the knapsacks in each round. As shown in the graph, it is still extremely competitive taking only 333 seconds even for the largest problem size 575 M (2500 applications on over a 1000 nodes). The good performance of CPA-R1 and the iterative nature of CPA suggest that it can in fact be terminated earlier after fewer rounds if a quicker decision is required, thus providing a smooth time-quality tradeoff.

4.4. Experiments with Increasing Tightness of Fit

The performance of any placement algorithm depends on how tight the packing and demands are. This is what we call the "tightness" of fit and can be controlled by the α parameter as described in §4.2. Increasing α leads to higher mean ($\frac{\alpha * N_c * cap(C)}{N}$) and hence tighter packings. It is easy to see that $\alpha = 1$ is the tightest packing and at high α values feasible solutions may not exist for any solver.

Figure-7 shows the results as α increases 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, requiring one to use far-off node pairs, resulting in a overall increased cost. Of the greedy algorithms, PAIR-GR performs much better than INDV-GR at lower α values wherein plenty of close-by unfilled compute-storage node pairs may be present and PAIR-GR is able to take advantage of them while INDV-GR is not. Comparatively, CPA continues to outperform other algorithms for all α values. And the relation of CPA-R1 is similar.

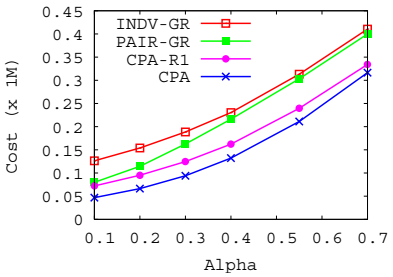

Figure 7. Quality with increasing tightness of fit

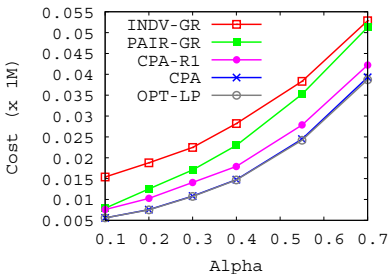

Figure 8. Comparison with OPT-LP

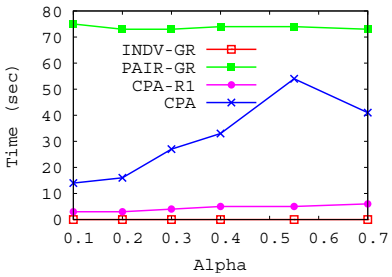

Figure 9. Time with increasing tightness of fit. CPA *used 3, 3, 5, 5, 7, 6 rounds*.

Figure-8 shows the same graph for a smaller problem size 0.14M (150 applications) where the OPT-LP was able to run to completion. Even here the trends are similar, with CPA pretty much overlapping the OPT-LP curve for all values of α. This illustrates the good optimizing quality of CPA with increasing tightness of fit.

Running Time Comparison: For processing time (Figure-9), the greedy algorithms are pretty much constant with increasing α values. So is the case with CPA-R1. The running time of CPA however varies with increasing α as it uses different numbers of rounds (between 3 and 7) for different αs. Since the termination for CPA happens upon reaching a local optimum, the number of rounds seem to differ for different αs. However, the good performance of CPA-R1 and the iterative nature of CPA imply that the algorithm can be terminated after fewer rounds (say 3) if time is a constraint and a fast answer is desired, thus exploiting the time-quality tradeoff.

4.5. Experiments with Increasing Variance

Similar to tightness of fit, a relevant parameter is the uniformity of the workloads. For example, if all workloads have same computation, storage requirements and same throughput rates, all solutions will tend to have the same cost as not much can be gained with different strategies during placements. As workloads become more diverse, there are different combinations that are possible and different algorithms exploit them differently.

To increase the variance among the workloads, we use the β parameter as described in §4.2. As the value of β increases, the standard-deviation ($\frac{\beta * N_c * cap(C)}{N}$) increases and the workloads become less uniform.

Figure-10 shows the performance of the algorithms as β is increased from 0 to 0.55 for a problem size of 37 M (1000 applications) with α=0.55 and DF=2. Notice that as β increases, the costs drop for all algorithms. This is because with the availability of many applications with low resource requirements, it is possible to fit more applications at smaller distances. However, among all the algorithms, CPA is able to react the best to this and achieve much better cost savings at higher β values. This is partly due to the knapsacks component of CPA. In the case of greedy algorithms, an early sub-optimum decision can cause poor fitting later on: for example, choosing a 600 item on a 800 capacity node prevents a later combination of 500 & 300.

Figure-11 shows the same experiment for a smaller problem size of 0.14M (150 applications) where OPT-LP was also able to run to completion. The results are similar with CPA providing solutions very close to OPT-LP.

Running Time Comparison: Figure-12 shows the rnning times for the different algorithms for the larger 37M problem size with increasing β. As before, greedy algorithms tend to maintain constant running time with increasing β and so is the case with CPA-R1. The running time of CPA however 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, the total time is still small and coupled with the good performance of CPA-R1 and iterative nature of CPA, it again suggests that it can be terminated earlier after fewer rounds (say 3) if a faster answer is desired, thus exploiting the time-quality tradeoff.

4.6. Experiments with Increasing Distance Factor

The last set of experiments vary the distance factor (DF) that is used to obtain distance values between computation and storage nodes. As mentioned earlier, the distance value is obtained using the formula DF^l where l is the number of physical inter-hop levels between nodes and is the same for all applications. 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.

Figure-13 shows the solution quality of the different algorithms as the distance factor is increased from 2 to 15 for a problem size of 37 M (1000 applications) and α=β=0.55. The CPA and CPA-R1 appear as if a single line due to their small relative difference compared to the scale of the graph. The most interesting, and in fact surprising observation is the performance of INDV-GR at higher DFs. It outperforms

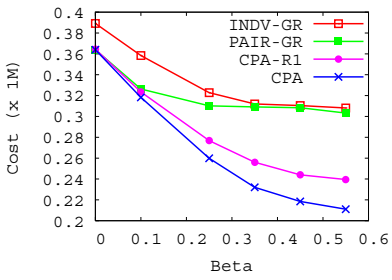

Figure 10. Quality with increasing variance

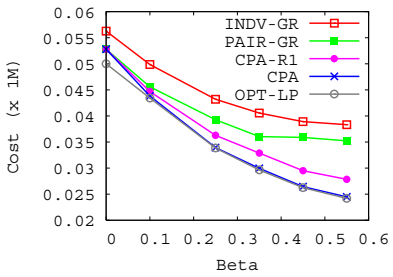

Figure 11. Comparison with OPT-LP

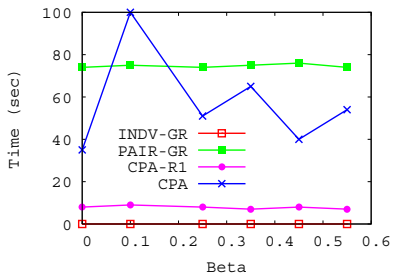

Figure 12. Time with increasing variance. CPA *used 3, 10, 6, 8, 5, 7 rounds*.

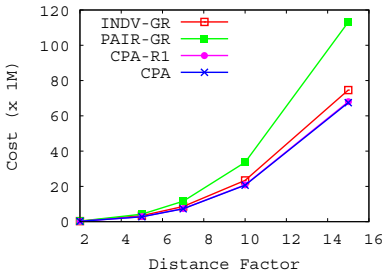

Figure 13. Quality with increasing distance factor. *INDV-GR fits better at higher DFs*.

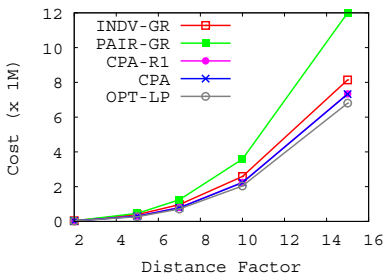

Figure 14. Comparison with OPT-LP. CPA *is still within 8% of OPT-LP at DF=15*.

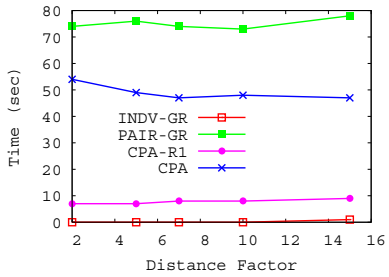

Figure 15. Time with increasing distance factor

PAIR-GR and comes reasonably close to CPA. On close inspection, we found that INDV-GR was performing well at placing applications at higher distances – i.e., even though it fit fewer applications and iorate at lower levels (l=0 and l=1), it fit more at l=3 and less at l=4. At higher DF values the contribution from higher levels tends to dominate the overall cost and hence getting that right is more important. We believe that this may lead to additional insights to further improve the performance of CPA for higher DFs.

Figure-14 shows the same graph for a smaller problem size of 0.14 M (150 applications). While the other trends are similar to before, one noticeable part here is that the OPT-LP is able to separate itself slightly more from CPA at higher DF values than at lower ones. Still, CPA remains within 8% of OPT-LP at $DF = 15$. This is partly due to the amplification factor as well. Finally, Figure-15 shows the running times of the different algorithms. The running time does not seem to vary much with changing DF confirming that it does not impact the fitting much.

4.7. Discussion

- From the evaluation, it is clear that CPA is **scalable** in optimization quality as well as processing time. It yields high quality solutions with speeds better by an order of magnitude or more. Secondly, the **iterative** nature of CPA allows improving any initial configuration, with an attractive property of further trading off computation time with solution quality.
- CPA 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.
- **Versatility of the framework**: The algorithms and the CPA framework allow the cost for each <application, server, storage> triple to be set independently. This allows the user to capture special affinities between application, compute and storage node triples by controlling the cost function. As an example of its versatility, the $Cost(A_i, C_j, S_k)$ may be of the form $Cost_{C_j} + Cost_{S_k} + \alpha * Affinity(C_j, S_k)$ where $Cost_{C_j}$ represents the cost of using C_j server (similar for S_k) and α weighing the importance of affinity between C_j and S_k. Higher the α, greater the need for coupled placement like CPA.

Other useful characteristics and extensions to the CPA framework including a deployment architecture are discussed in [35].

5. Conclusions and Future Work

In this paper we presented a novel algorithm, called CPA that optimizes coupled placement of application computational and storage resources on nodes in a modern virtualized data center environment. By effectively handling proximity and affinity relationships of compute and storage nodes, CPA 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 makes it especially suitable for dealing with dynamic scenarios like workloads surges, scheduled downtime, growth and node failures [35]. This work is part of a broader virtualization management project called SPARK which aims at managing server and storage virtualized resources in data centers in an integrated manner for different stages of virtualization lifecycle including P2V consolidation [18], provisioning and dynamic load-balancing [36].

CPA's good optimization quality and foundation in two well-studied theoretical problems seems to indicate a potential deeper theoretical connection and is an interesting area for future analysis. Other interesting area is to consider higher dimensionality while placing resources as well as newer cost models for other data center objectives like power and energy utilization.

References

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

[2] CPLEX 8.0 student edition for AMPL. http://www.ampl.com/DOWNLOADS/cplex80.html.

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

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

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

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

[7] Tiburon project at IBM almaden.

[8] Cisco MDS 9000 Family SANTap service - enabling intelligent fabric applications, 2005.

[9] Data Center of the Future, IDC Quarterly Study Number 06C4799, April 2006.

[10] VMware infrastructure 3 architecture, 2006.

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

[12] 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.

[13] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[14] 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.

[15] 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.

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

[17] 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.

[18] M. Cardosa, M. Korupolu, and A. Singh. Shares and utilities based power consolidation in virtualized server environments. In *Proceedings of IFIP/IEEE Integrated Network Management (IM)*, 2009.

[19] 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 Symposium on Operating System Principles*, 2001.

[20] 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.

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

[22] C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, 2005.

[23] L. W. Dowdy and D. V. Foster. Comparative models of the file assignment problem. *ACM Surveys*, 14, 1982.

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

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

[26] 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.

[27] 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.

[28] D. G. McVitie and L. B. Wilson. The stable marriage problem. *Communications of the ACM*, 14(7):486–490, 1971.

[29] M. Nelson, B. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *USENIX Annual Technical Conference*, pages 383–386, 2005.

[30] D. Pisinger. A minimal algorithm for the 0-1 knapsack problem. *Journal of Operations Research*, 45:758–767, 1997.

[31] 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.

[32] 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.

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

[34] D.B. Shmoys and É. Tardos. An approximation algorithm for the generalized assignment problem. *Journal of Mathematical Programming*, 62(1):461–474, 1993.

[35] A. Singh, M. Korupolu, and B. Bamba. SPARK: Integrated Resource Allocation in Virtualization SAN Data Centers. In *IBM Research Report RJ10407*, 2007.

[36] A. Singh, M. Korupolu, and D. Mohapatra. Server-storage virtualization: Integration and load balancing in data centers. In *Proceedings of IEEE/ACM Supercomputing*, 2008.

[37] A. Verma, P. Ahuja, and A. Neogi. pMapper: Power and Migration Cost Aware Application Placement in Virtualized Systems. In *ACM Middleware Conference*, 2008.

[38] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings of Symposium on Networked Systems Design and Implementation)*, 2007.