Vector Repacking Algorithms for Power-Aware Computing

Mario E. Consuegra, Giri Narasimhan, Raju Rangaswami
School of Computing and Information Sciences,
Florida International University,
Miami, FL 33140, USA.
E-mail: {mcons004,giri,raju}@fiu.edu

***Abstract*—In this paper we experiment with practical algorithms for the *vector repacking* problem and its variants. Vector repacking, like vector packing, aims to pack a set of input vectors such that the number of bins used is minimized, while minimizing the changes from the previous packing. We also consider a variant of vector repacking that stores additional copies of items with the goal of improving the performance of vector repacking algorithms. In addition, our algorithms are parameterized so that they can be effectively optimized for a variety of resource allocation applications with different input characteristics and different cost functions.**

I. Introduction

Data centers can consume up to a hundred times more energy than a standard office building, much of which is unfortunately wasted because they draw 60% of their peak power demand even when they are doing nothing. Power consumption by data centers has displayed phenomenal growth in recent years and is expected to grow at even faster rates [9], [8]. It is therefore imperative for storage systems to be more energy efficient in order to reduce unnecessary energy expenditure, save money, and cut down our carbon footprint.

Most data centers are provisioned for peak performance, while average loads can be much more modest. Toward the goal of lowering data center power consumption, Barroso and Hozle [11] have proposed the approach of "energy proportionality". An energy-proportional approach allows for power consumption to vary with the usage. This approach allocates the maximum sustainable amount of work into the minimum number of servers possible, while powering down the unused servers.

Resource allocation problems can be cast as vector packing problems by representing each item (task) and each bin (server) as a multi-dimensional vector with dimensions for relevant parameters. (We refer to the vector representing the item or bin as its *profile*.) In the storage systems application, relevant parameters of the profile include working set size, workload intensity (measured in IOPS), miss rate, etc. In the VM management application, the dimensions could represent the requirements of the VM for resources such as CPU, memory, bandwidth, etc. Energy-efficient computing can be achieved by considering the problem of (vector) packing these items into bins (servers) with the goal of optimizing (minimizing) the total power consumed by the servers. Server capacities in each of the cases directly translates to bin capacities for the abstracted multi-dimensional vector packing problem.

With the goal of optimizing the power consumption of data centers, Amur et al. [1], Verma et al. [17] and Thereska et al. [15] considered the problem of storing multiple replicas of data sets and working sets. Also, Panigrahy et al. [13], looked at the problem of efficiently packing virtual machines (VMs) with known static demands into servers with fixed capacitites. Using fewer bins (i.e., servers) directly translates to lower power consumption. Energy efficiency is thus achieved by keeping an optimal subset of servers in the system active while other servers are spun down or brought to a lower energy consumption state.

a) Vector Repacking: Given that workloads are inherently dynamic [11], we turn our attention to the challenging vector *repacking* problem. While vector packing strategies can calculate efficient packings that use close to the minimum number of bins, these packings may become sub-optimal as conditions change. But computing good new packings from scratch and repacking all items accordingly may cost a prohitively expensive amount of movement. Here we analyze multi-dimensional vector packing where we are required to "repack" efficiently and where the cost of the resulting packing is also dependent on how it differs from the previous packing, the assumption being that repacking requires a costly "migration" of tasks, processes, or data. Our approaches are meaningful only when the vectors to be packed change their profiles relatively infrequently, thus making it worthwhile to reconsider the repacking of the entire set of tasks.

b) Vector Repacking with Replicas: Next, we consider a practical variant of the vector repacking problem.

In this variant, we assume that the system can store a limited number of extra copies (replicas) of select (or all) tasks, with the goal of reducing the cost of "migration". Here our experiments aim to *study the tradeoff between overprovisioning and the cost of task/data migration*.

In sections III and IV we introduce solutions to the two problems described above and analyze them experimentally. The results show that our solution for vector repacking is an effective and practical approach to deal with problems from the areas of dynamic resource allocation and power-aware computing. We show that allowing for extra copies (replicas) of the entities can be used with vector repacking approaches to find efficient solutions that attempt to minimize migration costs. We show results of our experiments with some real data sets (Section V). Our conclusions are summarized in Section VI.

II. Related Work

The purpose of this section is to evaluate and select competitive (static) vector packing algorithms that could be useful to solve the dynamic repacking variants introduced above. As mentioned earlier, we model the problem of optimal placing of items such as working sets or VMs ("items") on disk servers or VM hosts ("bins") as a multi-dimensional vector packing problem. Efficient packings into the smallest number of bins translates to important energy savings. We assume that all bins have homogenous capacities in all dimensions and that the input is normalized such that the bins have a capacity of 1 in each dimension.

The multidimensional vector packing problem has been of interest for over three decades [12], [10] and many sophisticated approximation algorithms have been proposed for it. Even for $d = 1$, the vector packing problem (bin-packing) is NP-hard, and there are no approximation algorithms having an approximation ratio of $(\frac{3}{2} - \epsilon)$ for $\epsilon > 0$ unless $P = NP$ [3]. Hence finding efficient and practical algorithms to solve this problem is still a challenging task. For an excellent compilation of the relevant work on 1-dimensional bin packing the reader is referred to a survey by Coffman et al. [7].

For multi-dimensional vector packing Woeginger [18] ruled out an approximation scheme even for the case $d = 2$. For approximation algorithms, de la Vega and Lueker [6] obtained a $(d + \epsilon)$-approximation algorithm; and an improved algorithm by Bansal et al. [2] achieved an approximation ratio of $(\ln d + 1)$. Both algorithms are deemed not practical. For the 2-dimensional case, there is a $(\frac{1}{1-\rho})$-approximation algorithm (for any $\rho < 1$) called Hedging [5] (where ρ is the maximum length in any component for each item). Note that Hedging can be competitive and useful only for small ρ (e.g., $\rho < \frac{1}{2}$), but much simpler heuristics like First Fit perform as well with small values of ρ. Other relevant algorithms include generalizations of the most effective algorithms for the 1-dimensional case (e.g., GFFD), and were considered for the multi-dimensional vector problem considered here. Applications of vector packing to resource allocation problems have also been recently explored [13]. Lot less research has been done on the vector **repacking** problem. In fact, we are not aware of any relevant work on the two variants considered in this paper.

We assume that the input to all variants of the vector packing problem includes a set of d-dimensional vectors within the unit d-dimensional cube, representing the d-dimensional profiles of n items that need to be packed. In other words, $\bar{v}_i = (v_{i1}, v_{i2}, \ldots, v_{id})$, where $v_{ij} \leq 1$ for $j = 1, \ldots, d$. The (static) vector packing problem is to partition S into a minimum set of subsets of S (bins), $\{S_1, S_2, \ldots, S_m\}$, such that $\sum_{\bar{v}_i \in S_k} \bar{v}_i \leq \bar{1}$, for $k = 1, \ldots, m$, where $\bar{1}$ is the d-dimensional vector of all 1's.

Based on prior work, there is strong evidence that *off-line* algorithms for (static) vector packing perform better (both theoretically and in practice) than their *on-line* counterparts. Thus for the 1-dimensional case, First-fit decreasing (FFD) performs better than First-fit (FF). Similar behavior has been observed for vector packing in higher dimensions. For practical vector packing in 1-dimensions, it is well known that FF and Best-fit (BF) [7] and their off-line counterparts, FFD and BFD, strike the best balance between their time complexity and performance in terms of number of bins. Stillwell et al. [14] showed FFDSum to be a good choice for resource allocation algorithms for virtualized service platforms. Generalizing BF and BFD for vector packing can be done in many ways. For $d \geq 2$, Panigrahy et al. [13] proposed an algorithm called FFD-EL2, which finds the bin with the closest L_2-distance between the vector profile of the item and the remaining space in the bin (represented as d-dimensional vector). Their experiments showed FFD-EL2 to be the most competitive among the vector packing algorithms. In summary, generalizations of FF and BF for higher dimensions seemed to be the best candidates for applying to the vector repacking problems. However, our experiments led to some surprising results as shown below.

III. Vector Repacking

In practice, finding optimal or near-optimal placements of entities on servers is not the end of the story. When profiles of entities change, placements have to be modified, resulting in costly data migration between servers. Vector repacking is the problem that requires the simultaneous optimization of the number of bins as well as the amount of changes from a previous packing (i.e., migrations). The migration cost is modeled as a function of the difference between the previous packing and a new packing. More formally, we have:

Instance: A set of vectors $S = \{\bar{v}_1, \ldots, \bar{v}_n\}$, representing the d-dimensional profiles of the n items to be repacked, a partition of S, $B = \{S_1, \ldots, S_m\}$, representing the previous packing, and a cost function $f : (B, B') \to \mathbb{R}_{(0,1]}$ that assigns a cost to the change of packing from B to B'.

Problem: Find a packing $B' = \{S'_1, \ldots, S'_m\}$ such that $f(B, B')$ is minimized.

Note that we make the following assumptions. We assume that the cost function $f(B, B')$ is a combination of two costs – the cost of the packing B' (i.e., it depends on the number of bins in B') and the migration cost. We assume that the cost of migrating a specific item depends on its *size*, which is assumed to be one of the dimensions of vector profile representing the item (say, dimension 1). For example, the cost of migrating VMs is proportional to its memory footprint, while in storage systems, migration costs are proportional to the size of the working-set. We assume that the total migration cost for the whole repacking is simply the sum of the migration costs of individual items. In other words, other consequences of the migration (e.g., loss in computational time of a VM during the migration) are assumed to have minimal impact and negligble cost. Finally, we assume that the migration cost for a specific item depends only on the vector describing the item and not on the source or the destination of the move. For the applications in question, these assumptions are quite reasonable.

c) Applications: For resource allocation applications where one would like to assign tasks to servers, it is possible that a task may need to be migrated to a different server because its profile may change over a period of time. For example, a VM may become more or less compute intensive or memory intensive; in storage systems, a workload may have the miss rate characteristics change [17]. In the analogy of vector packing, it is possible that the bin may not be able to pack the same set of vectors as the parameters of the entities change, requiring migration of the entities from one bin (server) to another [16]. Here we run experiments simulating the scenario where a given set of data items is represented by a set of d-dimensional vectors. Our algorithms assume an initial packing for the items (by applying one of the static vector packing algorithms). Then as their profiles change the given placements may become untenable and new placements may be required, which involves data movement, whose cost is assumed to be proportional to the size of the migrated item. In the following experiments we study algorithms that aim to find a packing of data sets into a (approximately) minimum number of servers as the load varies over time while incurring a (approximately) minimum migration cost.

A. In Search of the Best Vector Repacking algorithms

d) The Repack *Algorithm:* For the vector repacking problem, the *naive* algorithm is to simply repack from scratch without using any information from the prior packing. The problem with the naive approach is that it can lead to a prohitively large movement cost. Smarter heuristics must try to incur less movement cost to achieve new good packings. We propose a generalized heuristic that can be easily and dynamically tuned to accomodate different user demands and trade-off choices. This generalized heuristic for vector repacking has the following three stages:

- The **first stage** involves vector eviction, where bins whose capacities have been exceeded are identified and selected vectors are evicted.
- The **second stage** involves placement of the evicted vectors, which are packed either into one of the existing bins with adequate resources or into new bins.
- The **third and final stage** involves a packing reduction step, where the entire contents of bins are considered for repacking into other bins; bins emptied in this manner are then closed.

The last stage takes care of consolidating underutilized bins where the goal is to see if the bin could be done away with entirely. As mentioned earlier, in our experiments, the total data movement cost of these operations was simply modeled as the sum of the first dimension of all the relocated items (i.e., size dimension).

Repack requires the choice of a (static) vector packing algorithm, which is then applied to obtain an initial packing of the items; in subsequent *intervals* it again uses a vector packing algorithm to decide how to evict items from overloaded and under-filled bins, and uses it again to decide where to repack them. We tested *Repack* using the off-line version in which all items to be packed are preordered in decreasing order of sum of all dimensions. We measured the quality of the algorithm in terms of number of bins and migration cost. Note that the migration cost is a function of the total size of all the items to be repacked.

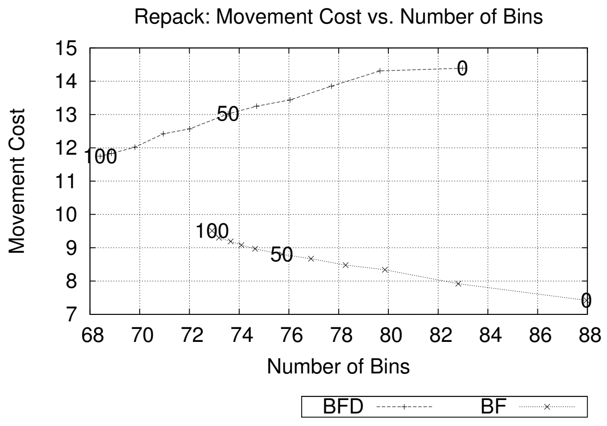
For simple packing, previous work had already shown that all the versions of BF and FF have comparable performance in practice. Along with the versions of BF, we also implemented Next-fit (NF). The results on the number of bins were exactly as expected, i.e., BF resulted in far fewer bins than NF. However, we made a curious yet crucial observation. We noticed that NF resulted in considerably less migration cost than BF. If the number of bins were the only criterion, then BF and FF are clear winners. Since in the dynamic setting the cost is measured in both the number of bins used as well as the migration cost, it means that the NF algorithm has some merit. Thus, BF and NF represent two extremes – NF is a fast algorithm and produces packings that are more expensive in terms of the number of bins used, while BF is a slower $O(n \log n)$ algorithm and achieves better packings. Note that the Worst-Fit (WF) algorithm and NF have been shown to have the same worst-case performance ratio. The NF packing algorithm has a smaller running time than WF and hence offers a better option as an extreme deviation from BF than WF can offer.

In the dynamic case achieving very good packings produces allocations that can become infeasible with higher probability once the values of the items change over time. NF has the advantage of incurring relatively low migration costs. The observation could be explained by the fact that because NF does not generate tight (efficient) packings when some of the items change their profiles there is higher probability that the bins will continue to have enough capacity to hold them since they have more slack. The result is that there is less need to migrate, and thus the migration cost is reduced. The take-home message was that *tighter packings will lead to higher migration costs*.

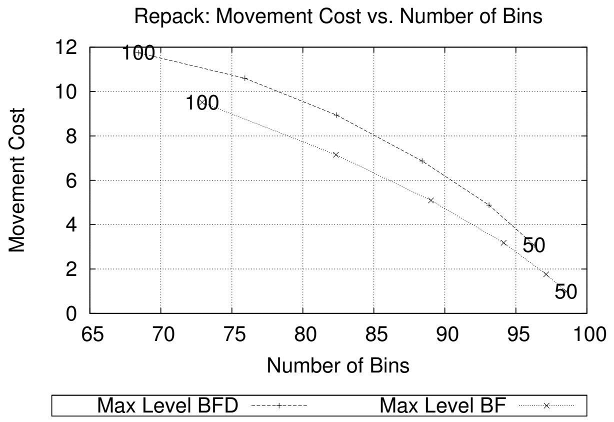
For different applications of the vector repacking problems there may be different acceptable trade-offs between the movement cost and the packing cost. Furthermore the dimensions of the items being repacked might follow different distributions in different situations. In order for *Repack* to be tunable and easily adaptable to different situations we must be able to emphasize movement cost or packing cost as needed for each different scenario in order to minimize cost. For this we need a parameter that can allow the *Repack* algorithm to target different levels of "tightness" of the packings. We identify two candidate parameters for this. One of them is to limit the amount of choices of open bins for each item when applying vector packing heuristics. The other one is two impose a slack space on each bin.

e) The Hybrid Vector Packing Algorithm: We propose two families of algorithms. First, we propose a family of algorithms which we will refer to as k-bounded BF (on-line) and k-bounded BFD (off-line), for different values of k. We abbreviate these algorithms as kBF and kBFD. NF can be described as an algorithm that looks at *one* bin and decides whether or not to place the next item in it (or to open a new bin). In contrast, BF can be thought of as an algorithm that looks at *all* open bins and decides on the best choice of a bin where the item is to be placed (or opens a new bin). The k-bounded BF looks at a fraction k of the bins to decide on the best choice of a bin to place the next item. When $k = 0\%$, the algorithm is same as NF and when $k = 100\%$, i.e., equals the number of bins currently open, the algorithm is the same as BF. The second family of algorithms we propose will be referred to as k-MaxLevel BF (online) and k-MaxLevel BFD (offline). These vector packing algorithms fill each bin up to no more than k percent of their capacities along each dimension.

We studied the use of kBF, kBFD, k-MaxLevel BF and k-MaxLevel BFD with *Repack* to characterize the effect of the packing algorithm on the trade-off between the migration cost of repacking at each dynamic interval and the number of bins used. We tested *Repack* with kBF and kBFD for $k = 0\%, 10\%, 20\%, \ldots, 100\%$. and *Repack* with k-MaxLevel BF and k-MaxLevel BFD for $k = 50\%, 60\%, \ldots, 100\%$



(a)



(b)

Fig. 1: *(a) The effect of varying k on the performance of kBF and kBFD. Note that $k = 0\%$ and $k = 100\%$ correspond to NF and BF respectively. (b) The effect of varying k on the performance of kMaxLevel BF and kMaxLevel BFD.*

Figure 1 highlights the tradeoff between the packing efficiency and the migration cost as k is varied. Here we tested *Repack* with its kBF, kBFD, kMaxLevel BF, and kMaxLevel BFD versions on 100 test cases each with 100 items, 4 dynamic intervals and values from the uniform distribution $U(0, 1]$. The first observation is that the tradeoff is non-existent for the kBFD version but is manifested in the kBF, kMaxLevel BF and kMaxLevel BFD versions. With kBFD both the number of bins and the migration cost go down with increase in k. The behavior of the kBFD version, for which in each vector packing operation the input is sorted in decreasing order of some measure of the item, is explained as follows. As k increases, it is no surprise that the number of bins becomes smaller since every item has more choices of bins to be placed. (In fact, the reason for the decrease in the number of bins also applies to the on-line case.) For the kBFD, at the start when the items are "large", the bins are packed loosely. Then because the items are packed in decreased order of size then the last bins to be opened contain more items than the bins that were opened earlier, since the bigger items are placed in the first bins to be opened and take more space. The smaller items are packed together more tightly and in larger numbers than the larger items with the kBFD algorithm. When the items's demands change, the bins that contain more items will overflow with higher probability. As k increases, the algorithm packs the last k bins less tightly (since it has more choices) causing lower migration costs from these k bins in the next interval. We have already argued that as bins get packed less tightly, the migration costs start to drop. In the on-line case, since the items appear in random order, all bins are packed to roughly the same level of tightness. Therefore, for the on-line case, fewer bins translates very simply to greater migration costs. Our experiments (see Figs 15 and 16 in the Appendix) support the above arguments. Another interesting observation is that the biggest drop in number of bins happens when we go from $k = 0\%$ to $k = 10\%$. This is not surprising since having a few choices is always better than no choices. However, the marginal gain of more choices starts to drop as the number of choices increases.

The real merit of on-line kBF is that it is possible to "exploit the tradeoff" and optimize the cost of the algorithm by picking the value of k that balances the packing cost and the migration cost, thus providing finer control on the combined total cost.

Using kMaxLevel BF and BFD with *Repack* shows even a finer relationship between the movement cost and the packing cost as k is changed. In contrast to kBF and kBFD, the MaxLevel versions keep all bins open, and hence all bins are packed more evenly, so the packings produced are less likely to overflow once the items change profile since the load is likely to increase more evenly among all bins (see Fig 17 in the Appendix). One disdavantage of the kMaxLevel versions compared to kBF and kBFD is that since kMaxLevel looks at more choices than kBF and kBFD for smaller k it takes more time to compute a new packing. These behaviors stay consistent with tests in higher dimensions (see Figure 13 and 14 in the Appendix).

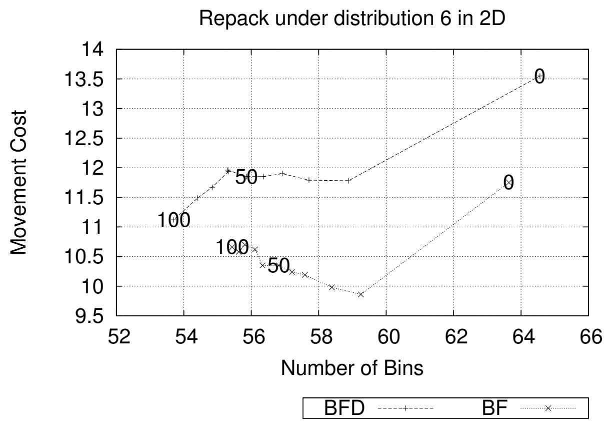
f) Experiments with Other Distributions:: As pointed out by Panigrahy et al. [13], practical resource allocation applications vary widely in terms of how heterogeneous they are in their resource requirements. It is therefore important to perform comprehensive experiments with different distributions under a variety of correlations across dimensions. The following set of experiments were inspired by the work of Panigrahy et al. [13] and Caprara and Toth [4].

The *Repack* algorithm was tested on 100 different simulated data sets each with 100 items. In each test case, the initial vectors were randomly generated using values from $U(0, 1]$ and distributions from Caprara and Toth [4]. Then for each of 4 *intervals* the values of these vectors were changed in all dimension (except the first dimension, which we refer to as the static size dimension), again generating values from different uniform distributions. This is consistent with what is likely to happen in storage systems where working sets do not change significantly in size over a small time *interval* [17]. We use the following 8 interesting distributions from Caprara and Toth [4]. Distributions *Caprara 1* through *6* correspond to $U[0.1, 0.4]$, $U[0, 1]$, $U[0.2, 0.8]$, $U[0.05, 0.2]$, $U[0.025, 0.1]$, and $U[0.133, 0.667]$, respectively. Distribution *Caprara 7* corresponds to $U[0.133, 0.667]$ for odd dimension i and to $U[v_i - 0.067, v_i + 0.067]$ for dimension $i + 1$, where v_i is the value sampled for dimension i. Lastly, *Caprara 8* corresponds to $U[0.133, 0.667]$ in dimension i and $U[0.733 - v_i, 0.867 - v_i]$ in dimension $i + 1$. Thus distributions Caprara 7 (and 8) correspond to a positively (resp., negatively) correlated distributions.

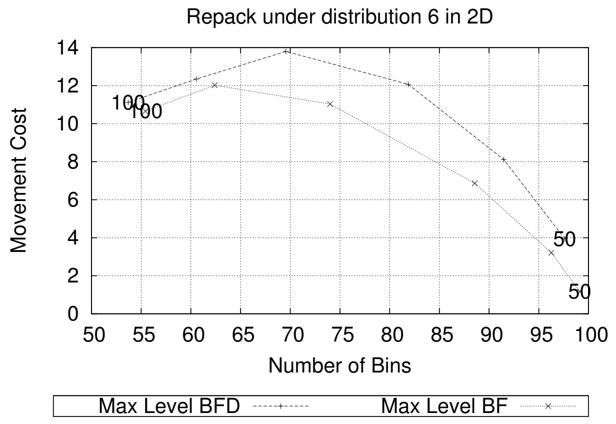
The behaviors observed under some of the above distributions suggest that the choice for best version of *Repack* can change drastically from one distribution to the other. In Figure 2, which shows the results on the *Caprara 6* distribution, we can see that if the objective was to minimize the movement cost while keeping the number of bins under 60 bins then the kBF version of *Repack* with $k \approx 10$ would be the best choice. Results in the rest of the distributions can be seen in the Appendix (see Section VII-A).

IV. Vector Repacking with Replicas

In many resource allocation applications (e.g., storage management and VM management), migration of tasks is often prohibitively expensive. In the previous section we saw how to compensate for it by picking the right value of k in the kBF packing algorithm. Another way to lower or even eliminate data migration costs is to replicate and over-provision the resources at the start [17]. The storage of redundant copies of the items on different servers is a technique already used in storage systems for achieving higher fault tolerance and robustness. (However, here we argue that it also results in lower migration costs.) For example, in Verma et al. [17], the system selects



(a)



(b)

Fig. 2: *Performance of* Repack *with input vectors from the distributions (a)* Caprara *6 with kBF and BFD and (b)* Caprara *6 with kMaxLevel.*

one of the copies of each item to be its *active* copy. Other copies remain on different servers as inactive. When a server exceeds its capacity, then the system may select one or more of the items whose active copy is on that server, and deactivate it. For each of the deactivated copies, the system then has to choose one of the inactive copies (on a different server) and make it active, thus effectively shifting the load corresponding to that item from one server to another. However, this does not mean that the migration cost can be totally eliminated. When an item switches to a different active copy then synchronization of the copies may be needed since only the active copy is involved in computation and may have undergone some changes. Synchronization is the process of ensuring that the copies are consistent. In the worst case, synchronization costs can be as high as migration costs. However, it is estimated that synchronization costs are on the average considerably lower than migration costs. Thus the replication approach, attempts to tradeoff overprovisioning, replication, and copy synchronization costs with that of migration costs.

We considered the effect of over-provisioning resources to store replicas of entities with the goal of lowering data migration costs in the event that parameters (such as workload intensity) of entities change. We developed an algorithm called *Replicas* that creates multiple copies (replicas) of each item and strategically places the copies on different servers. When the profiles of the entities change, the algorithm adapts by just selecting a different active copy of the multiple copies without moving the item to a different location. The synchronization associated with switching the active copy involves reconciling the fraction of the item "dirtied" since the last update of that replica.

A. *The* Replicas *Algorithm*

The *Replicas* algorithm assumes the choice of a vector packing heuristic used in assigning tasks/items to servers. It consists of two parts: REPLICA ALLOCATION (see Algorithm 1), and ACTIVE REPLICA SELECTION (see Algorithm 2). Given the ordering and packing strategies, and given α, the over-provisioning factor, REPLICA ALLOCATION makes repeated scans of the n items (using the order determined by the given ordering strategy) and places the replicas in $M(1+\alpha)$ bins using the given packing strategy, where M is the number of bins needed by that algorithm to pack one copy of all items. Note that the cost of Algorithm 2 is the cost of data movement involved in placing replicas in bins and the number of bins needed.

Algorithm 1 Replicas: REPLICA ALLOCATION

```
Require: $V$: list of $n$ entities;
  $\alpha$: over-provisioning factor;
  $A$: vector packing subroutine

  initialize $B$ to array of $n$ empty bins
  PackOneCopyOfAll($B, V, p, q$)
  add $\lceil \alpha \cdot M \rceil$ new empty bins to $B$
  while (there is space to store more replicas) do
    PackReplicas($B, V, p, q$)
  end while
```

Once replicas have been strategically placed as shown above, Algorithm ACTIVE REPLICA SELECTION is used to determine which of the multiple replicas of each item is to be the "active" replica. (All other replicas are made inactive and are left in standby mode for possible later activation.) Note that Algorithm ACTIVE REPLICA SELECTION is also invoked after significant changes in the resource need profiles of the items. In storage system applications, this would be done at periodic intervals or after significant changes in load intensities [17]. For a relatively modest cost of storing multiple replicas, the advantage of switching active replicas is that capacity constraints of servers can be achieved with minimal data movement. Note that replicas are placed strategically and the choice of active replicas is made in a manner such that servers with no active replicas can be "turned off" or put in lower energy states to reduce power consumption.

Algorithm 2 Replicas: ACTIVE REPLICA SELECTION

```
Require: $V$: list of $n$ entities; $\alpha$: over-provisioning
  factor; $A$: vector packing subroutine with $q$: ordering
  rule for items;
  $E = \phi$ // initialize set of emergency bins
  $O = \phi$ // initialize subset of active bins
  for all ($i \in V$ sorted according to $q$) do
    $C$ = set of active bins with a replica of item $i$
    SELECTACTIVECOPY$(i, C, A)$
    if (no such bin) then
      SELECTACTIVECOPY$(i, \overline{C}, A)$
      if ($b$ is such a bin) then
        $O = O \cup \{b\}$ //set bin $b$ as active
      else
        PACKSINGLEITEM$(i, E, A)$
      end if
    end if
  end for
```

Algorithm ACTIVE REPLICA SELECTION maintains a list of currently chosen active replicas. Bins/servers that contain at least one active replica from that list are called "active" bins/servers. Initially there are no active replicas or bins. Items are scanned in the order determined by the ordering rule q. If replicas of that item are found in active bins (C), then one of those replicas is chosen as the active copy using the criteria from the vector packing subsroutine, A. If none of the replicas are in active bins, then an attempt is made to find one from the set of inactive bins ($\overline{C}$). If none of the active or closed bins with replicas can accommodate the item, then the item is packed in one of the emergency bins using algorithm A, and the item and the bin are made active. Note that the total cost of Algorithm ACTIVE REPLICA SELECTION is the movement cost (data movement) needed to place replicas in emergency bins (equal to the aggregate size of all the items placed in emergency bins), the synchronization cost of items with changed active replicas, and the total number of bins used. The emergency bins are discarded at the beginning of the next ACTIVE REPLICA SELECTION application in the subsequent interval.

B. Experiments

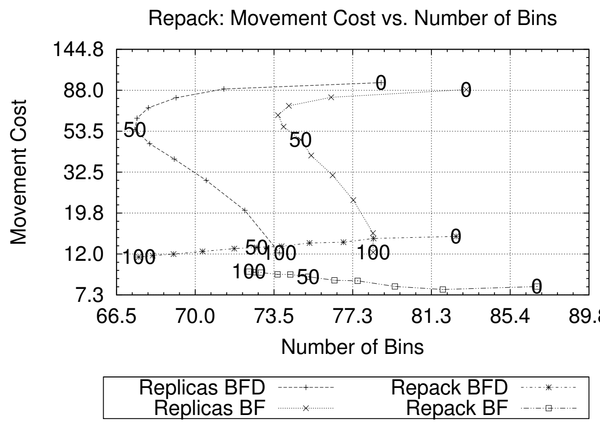
We tested the *Replicas* algorithm on 100 data sets and 100 items, each item represented by two-dimensional vectors with randomly generated values from $U[0, 1]$, and with 4 iterations of changes in the second dimension (load). Since switching between copies incurs a synchronization cost, we studied the viability and cost of replication with different dirty ratios. (*Dirty ratio* is the percentage of the item "dirtied" or modified since the last time a replica was made. In other words, it is the percentage difference between the original copy and its replica on another server.) As in *Repack*, we tested *Replicas* using kBF, kBFD, kMaxSizeBF, and kMaxSizeBFD versions of *Replicas*. As expected, higher *dirty ratio*s resulted in higher movement costs. We carefully examined the performance of *Replicas* with a moderate *dirty ratio* of no more than 12.5%.

Figure 3 (a) shows that a setting of $k \approx 30--50\%$ is the best option for *Replicas* kBFD and *Replicas* kBF. in terms of lowering the number of bins, while $k = 100\%$ was the best option for lowering the movement cost. Figure 3 (a) shows that for most values of k, *Repack* dominates *Replicas* in performance when $\alpha = 0$. (However, we will see later that the performance of *Replicas* can be improved by increasing the overprovisioning.) Fig. 3 (a) shows the number of bins decreasing as k grows from 0 to 50% and then increasing as k grows from 50 to 100%. This is caused by the large difference in the number of emergency bins used used by *Replicas* kBF and kBFD with different values of k, as shown in Fig. 4. When k is small these two variants of *Replicas* almost exclusively use emergency bins. Hence only when $k > 50\%$, we start to see more competitive behavior.

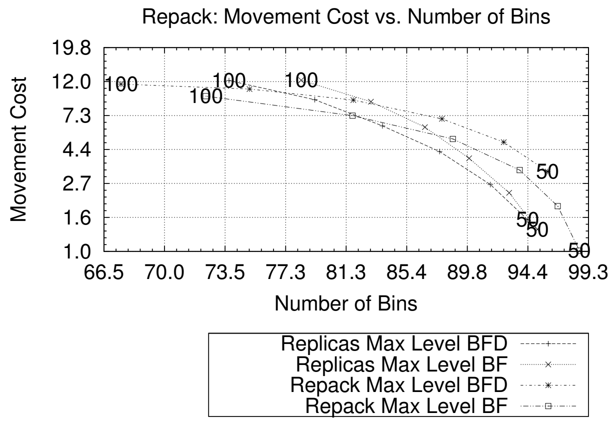
Fig. 3 (b) shows the kMaxLevel BF and BFD versions of *Replicas* and *Repack*. Here we can see that higher k results in better packings while incurring more movement cost, while lower k results in worse packing while decreasing movement cost. All the solutions shown in 3 (b) perform similarly for this case.

Fig. 5 shows the performance of *Repack* and *Replicas* with an overprovisioning factor of $\alpha = 1$. Note that the x-axis of the graphs represents the number of active bins, even though the number of bins employed to hold all replicas is $\lceil \alpha \cdot M \rceil$. *Replicas* takes advantage of the existence of copies to decrease movement cost, thus showing that the overprovisioning factor is an effective way to reduce movement cost. Fig. 5 (a) shows that for higher values of k, the movement cost reduces almost 5-fold, with only a minor increase in number of active bins. Fig. 5 (b) shows that the performance of the kMaxLevel versions of *Replicas* changes very little with respect to k, dominates the *Repack* versions for lower values of k, and exhibits almost a 5-fold reduction in movement cost for higher values of k.

An overprovisioning factor of $\alpha = 1$ gives *Replicas* its best performance in terms of movement cost. Increasing



(a)



(b)

Fig. 3: *Performance of the different versions of Replicas with overprovisioning factor $\alpha = 0$ and Repack for different values of k. Experiments used uniform distribution in 2D, and a maximum dirty ratio of 12.5%*

α more does not produce any visible gains over $\alpha = 1$. Smaller overprovisioning factors do not improve over $\alpha = 1$, though they still offer advantages over *Repack* for some cases (see Fig. 6 showing experiments with $\alpha = 0.75$).

V. Experiments with Real Trace Datasets

We finally tested our suite of algorithms on a storage systems application. Real trace data was generated from accesses to the university storage system. This enabled us to compare *Replicas* with the best existing algorithm called SRCMap to manage replicas in storage systems.

We used the same traces that were used to test the performance of SRCMap by Verma et al. [17]. The traces

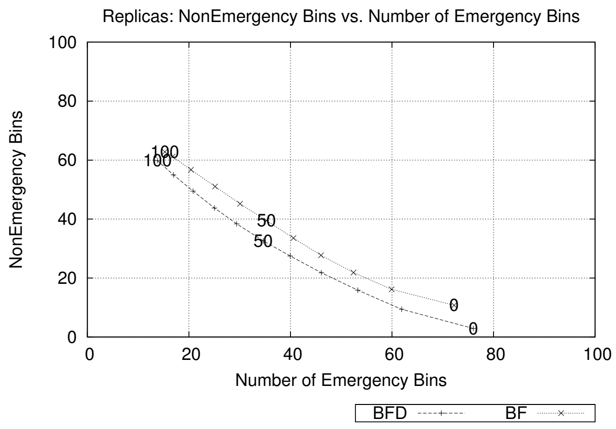
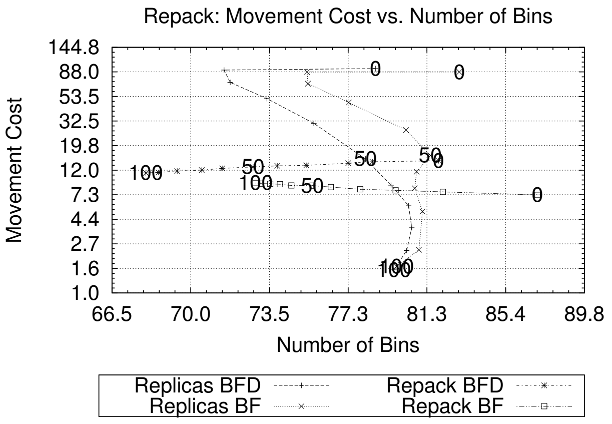


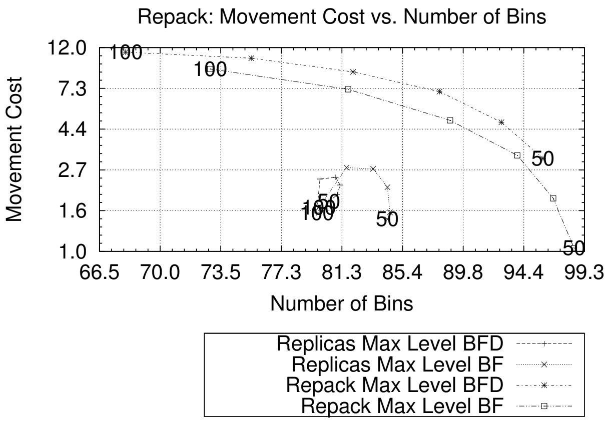
Fig. 4: *k-bounded Replicas with overprovisioning factor $\alpha = 1$ with kBF and kBFD. Experiments used uniform distribution and dirty ratio of 12.5%.*

included all I/O data requests over a period of 480 hours to 8 independent data volumes residing on 8 different disks. The algorithms we tested performed REPLICA ALLOCATION every 24 hours and ACTIVE REPLICA SELECTION every 1 hour long interval. The SRCMap heuristic also performed these two tasks at the same intervals of time. The traces were used to infer data about the average load intensities of each data volume over each period of one hour length. These values were then used as the realistic input that represents dynamic workload profiles of the data sets over each interval of one hour. Also for each data volume, the working set of the volume was determined by aggregating all data I/O requests to that volume.

We ran the experiments with 8 data volumes each with load capacity levels of $125, 196, 196, 196, 196, 196, 145$, and 145 IOPS respectively. Each data volume set aside 20% percent of its storage capacity as replica space to hold working set copies of other volumes. The data volumes had storage capacities of $270, 7.8, 7.8, 10, 10, 20, 170$, and 170 GBs respectively. Each disk $D_i \in [D_n]$ has a working set V_i. Every hour each $V_i \in [V_n]$ is assigned a new average workload value for the length of the interval. Every 24 hours $V_i \in [V_n]$ is assigned a new working set size value for the length of the 24 hours interval. Each working set V_i is replicated by storing its copies on multiple disks from $[D_n]$. Every 24 hours both *Replicas* and SRCMap apply their REPLICA ALLOCATION algorithm, and every hour they both apply their ACTIVE REPLICA SELECTION algorithm. *Replicas* was run with over-provisioning factor of $\alpha = 0$ for fair comparison to SRCMap which does not assume more than n servers. We measured the movement cost by finding the percentage of the total data that had to be

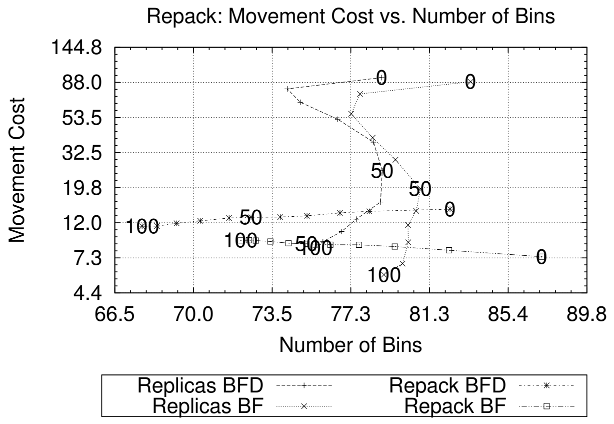


(a)

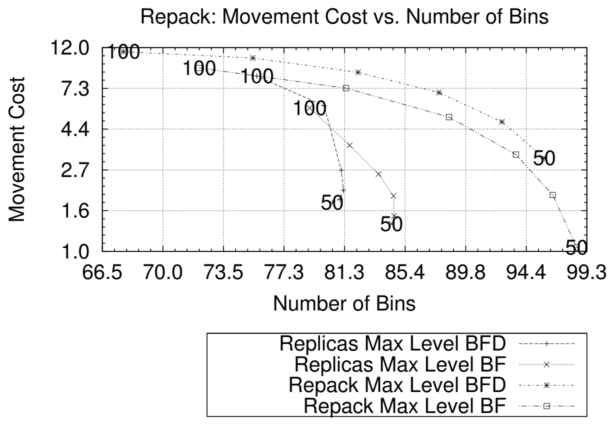


(b)

Fig. 5: *Comparing performance of k-bounded Replicas with overprovisioning factor $\alpha = 1$ and Repack for (a) kBF and kBFD and (b) kMaxLevel BF and kMaxLevel BFD. Experiments used uniform distribution and dirty ratio of 12.5%.*



(a)



(b)

Fig. 6: *Comparing performance of k-bounded Replicas with overprovisioning factor $\alpha = 0.75$ and* Repack *for (a) kBF and kBFD and (b) kMaxLevel BF and kMaxLevel BFD. Experiments used uniform distribution and dirty ratio of 12.5%.*

moved during the synchronization operations required when switching primary target for any working set V_i.

The SRCMap heuristic makes two assumptions that required us to modify *Replicas* in order to offer a more fair comparisson between the two solutions:

- Each working set V_i has its main copy stored in the primary storage section of disk D_i. Then each disk D_i serves all requests to V_i whenever it is active, even if another active disk D_j has a replica of working set V_i. Hence *Replicas* was modified so that in the ACTIVE REPLICA SELECTION stage of any item V_i it always selects the target D_i if this server is selected to be active.
- *SRCMap* assumes that in the worse case, each server D_i would be active and able to service the demands of its volume V_i. For this reason no emergency hosts were set up in *Replicas*.

We call this modified version of *Replicas* as *ReplicasWS*.

Table I compares the performance of *SRCMap* with versions of *ReplicasWS* using different vector packing (VP) strategies. These use k-bounded EL2 or EL2DSum. We show results for $k = 0, 50$, and 100. The movement cost of the *ReplicasWS* algorithms was much lower than that of SRCMap. The *ReplicasWS* versions using NF (labeled "ReplicasWS 0-EL2" in table I) kept active all the servers (total of 8) most of the time while incurring the lowest movement cost. The kEL2DSum and kEL2 variations of *ReplicasWS* for $k = 50$ and $k = 100$ incurred negligible movement cost while using almost half the number of bins as comapred to NF. The *ReplicasWS* algorithms with higher k used fewer bins on the average. In particular, *ReplicasWS* kEL2 with $k = 100$ used the least number of bins and was the only *ReplicasWS* algorithm to use fewer bins than *SRCMap* with a movement cost of less than half of that of SRCMap. These experiments suggest strongly that the best algorithm to pick can change with different distributions and objectives.

Algorithm	**Avg # Active Hosts ± Std Dev**	**Movement Cost ±Std Dev**
ReplicasWS 0-EL2	7.88 ± 0.94	0.07 ± 1.00
ReplicasWS 0-EL2DSum	3.71 ± 1.19	0.34 ± 1.00
ReplicasWS 50-EL2	3.14 ± 0.56	0.41 ± 2.00
ReplicasWS 50-EL2DSum	2.79 ± 1.05	0.44 ± 2.00
ReplicasWS 100-EL2	2.10 ± 0.86	1.08 ± 5.00
ReplicasWS 100-EL2DSum	2.47 ± 1.07	0.48 ± 2.00
SRCMap	2.35 ± 0.71	2.23 ± 8.21

TABLE I: Comparing *Replicas* to *SRCMap*

VI. Conclusions

This paper considers resource allocation problems with *dynamic* (but infrequent) changes in their profiles. Such problems have enormous practical relevance in this era of cloud computing where data centers and supercomputers store operate over large amounts of data and service large amounts of work incurring high energy costs.

Many important dynamic resource allocation problems can be cast as a vector repacking problem. We have proposed the *Repack* algorithm to address the vector repacking problem; it allows for "incremental" repacking to reduce the cost of migrations due to repacking.

We also considered a variant of the vector repacking problem that allows for a limited amount of extra bins to store multiple copies (replicas) of items. Experiments show that a small number of extra bins and multiple copies improves the performance in terms of packing efficiency as well as migration costs. Our experiments show that *Replicas* is a practical tool for resource allocation and power-aware computing for systems that store entities with dynamically (but infrequently) changing characteristics. All our experiments were confirmed with synthetic data sets produced from a collection of different realistic distributions, and with real traces from real systems.

Acknowledgments: This work was partly supported by NSF Grant CNS 1018262 and a NSF Graduate Research Fellowship DGE-1038321 for MEC. The authors thank Ricardo Koller and Luis Useche for providing the trace data sets used in our experiments.

References

[1] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan, "Robust and flexible power-proportional storage," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 217–228.

[2] N. Bansal, A. Caprara, and M. Sviridenko, "Improved approximation algorithms for multidimensional bin packing problems," in *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 697–708.

[3] N. Bansal, A. Lodi, and M. Sviridenko, "A tale of two dimensional bin packing," *Annual IEEE Symposium on Foundations of Computer Science*, vol. 0, pp. 657–666, 2005.

[4] A. Caprara and P. Toth, "Lower bounds and algorithms for the 2-dimensional vector packing problem," *Discrete Appl. Math.*, vol. 111, August 2001.

[5] S. Y. Chang, H.-C. Hwang, and S. Park, "A two-dimensional vector packing model for the efficient use of coil cassettes," *Comput. Oper. Res.*, vol. 32, pp. 2051–2058, August 2005.

[6] W. F. de la Vega and G. S. Lueker, "Bin packing can be solved within $1 + \epsilon$ in linear time." *Combinatorica*, pp. 349–355, 1981.

[7] J. E. G. Coffman, M. R. Garey, and D. S. Johnson, "Approximation algorithms for bin packing: a survey," in *Approximation Algorithms for NP-Hard problems*, D. S. Hochbaum, Ed. PWS Publishing Co., Boston, MA, USA, 1997, 1997, pp. 46–93.

[8] EPA, "EPA Report to Congress on Server and Data Center Energy Efficiency," U.S. Environmental Protection Agency, Tech. Rep., 2007.

[9] J. G. Koomey, "Worldwide electricity used in data centers," *Environmental Research Letters*, vol. 3, no. 3, 2008.

[10] L. T. Kou and G. Markowsky, "Multidimensional bin packing algorithms," *IBM J. Res. Dev.*, vol. 21, pp. 443–448, September 1977.

[11] U. H. L. A. Barroso, "The case for enrgy-proportional computing," *Computer*, vol. 40, pp. 33–37, 2007.

[12] K. Maruyama, S. K. Chang, and D. T. Tang, "A General Packing Algorithm for Multidimensional Resource Requirements," *International Journal of Computer and Information Sciences*, vol. 6, no. 2, pp. 131–149, 1977.

[13] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder, "Heuristics for vector bin packing." Microsoft Research, Tech. Rep., 2011.

[14] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova, "Resource allocation algorithms for virtualized service hosting platforms," *Journal of Parallel and Distributed Computing*, vol. 70, no. 9, pp. 962–974, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731510000997

[15] E. Thereska, A. Donnelly, and D. Narayanan, "Sierra: a power-proportional, distributed storage system," Microsoft Research, Tech. Rep., 2009.

[16] P. S. V. Sundaram, T. Wood, "Efficient data migration in self-managing storage systems," in *Proceedings of ICAC 06*, 2006, pp. 297–300.

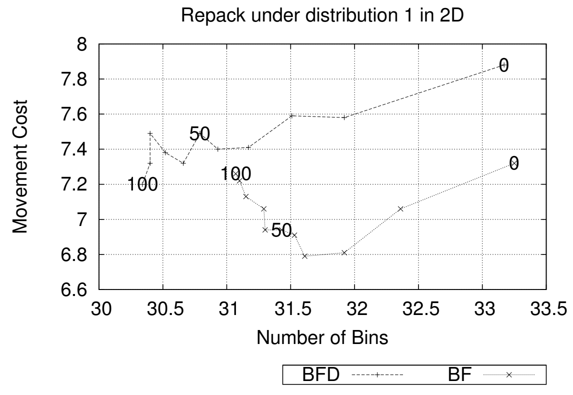
[17] A. Verma, R. Koller, L. Useche, and R. Rangaswami, "Srcmap: energy proportional storage using dynamic consolidation," in *Proceedings of the 8th USENIX conference on File and storage technologies*, ser. FAST'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 20–20.

[18] G. J. Woeginger, "There is no asymptotic ptas for two-dimensional vector packing," *Inf. Process. Lett.*, vol. 64, pp. 293–297, December 1997.

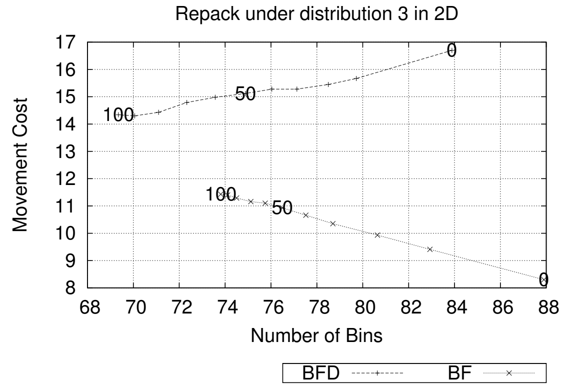
VII. Appendix

A. *Experiments with* Repack *on a variety of input distributions*

Fig. 7, 8, and 9 show the performance of *Repack* using kBF and kBFD under 6 different distributions: Caprara 1, 3, 5, 6, 7, and 8. The performance of *Repack* using kMaxLevel BF and kMaxLevel BFD under the same 6 distributions are presented in Fig. 10. 11, and 12.

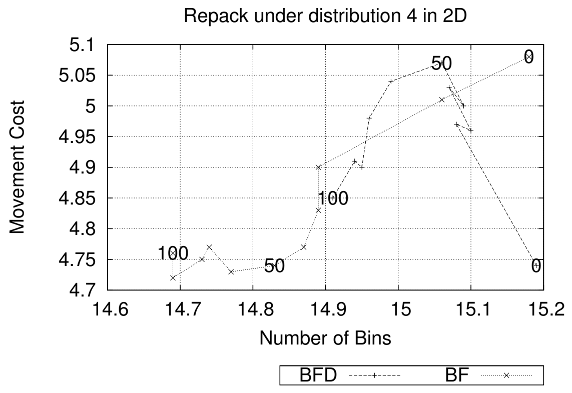


(a)

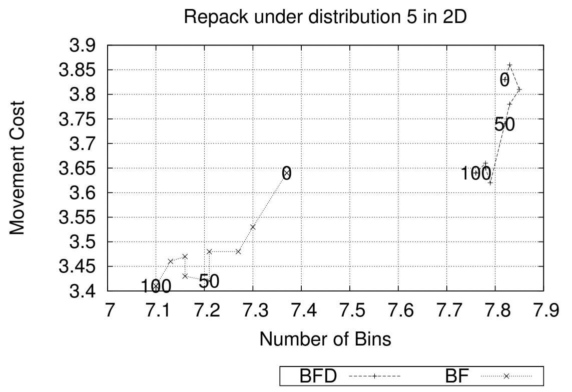


(b)

Fig. 7: *Performance of* Repack *using kBF and kBFD on distributions: (a) Caprara 1, (b) Caprara 3*



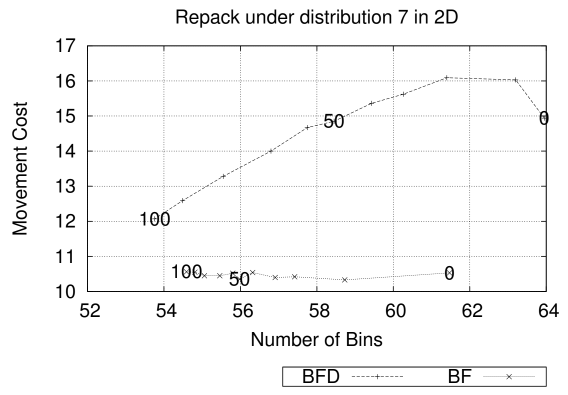
(a)



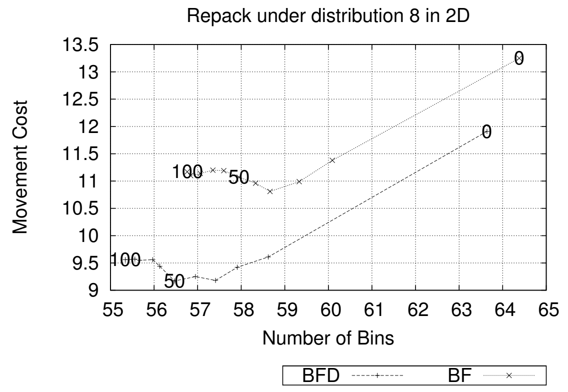
(b)

Fig. 8: *Performance of* Repack *using kBF and kBFD on distributions: (a) Caprara 4, (b) Caprara 5*

Fig. 13 shows the performance of *Repack* using kBF and kBFD with 4 and 8-dimensional inputs. Fig. 14 shows the performance of *Repack* using kMaxLevel BF and kMaxLevel BF with 4 and 8-dimensional inputs.

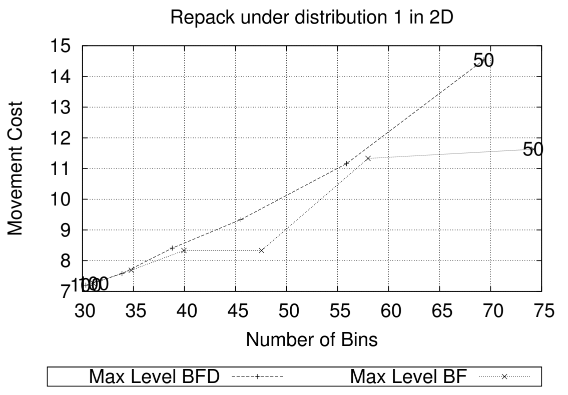


(a)

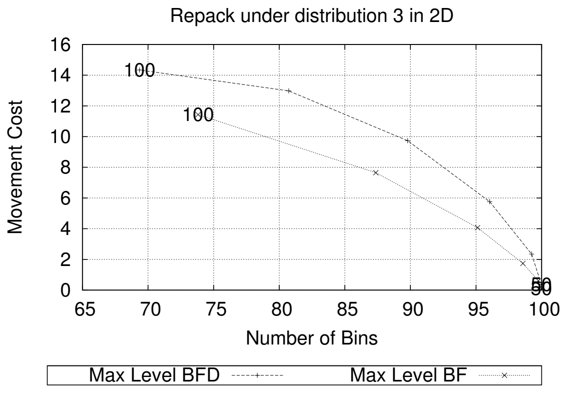


(b)

Fig. 9: *Performance of* Repack *using kBF and kBFD on distributions: (a) Caprara 7 (b) Caprara 8*

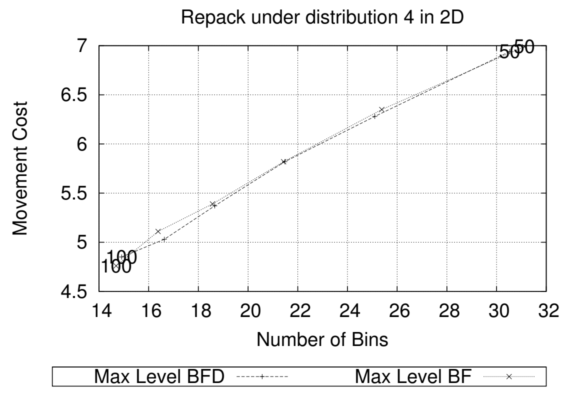


(a)

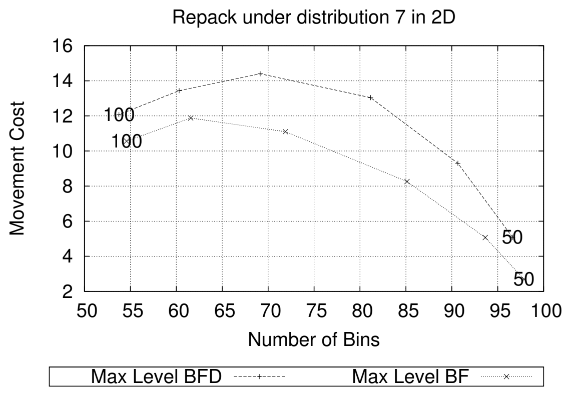


(b)

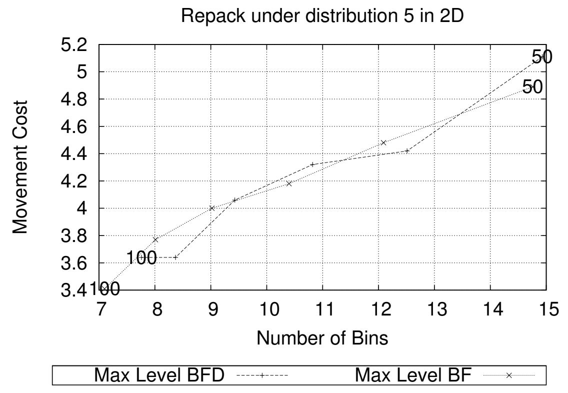
Fig. 10: *Performance of* Repack *using kMaxLevel BF and kMaxLevel BFD on distributions: (a) Caprara 1, (b) Caprara 3*



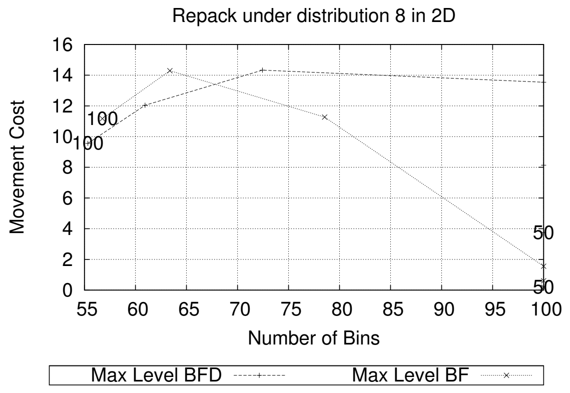
(a)



(a)



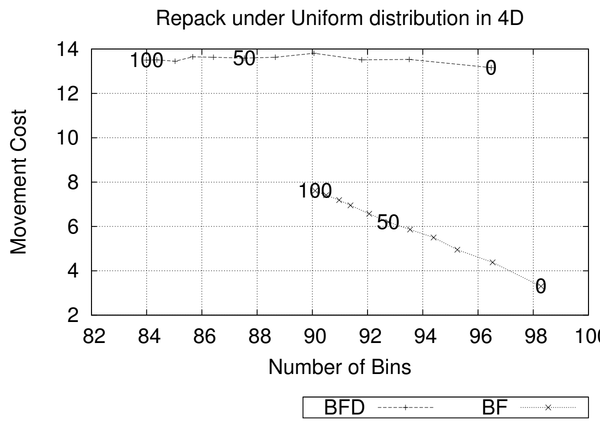
(b)



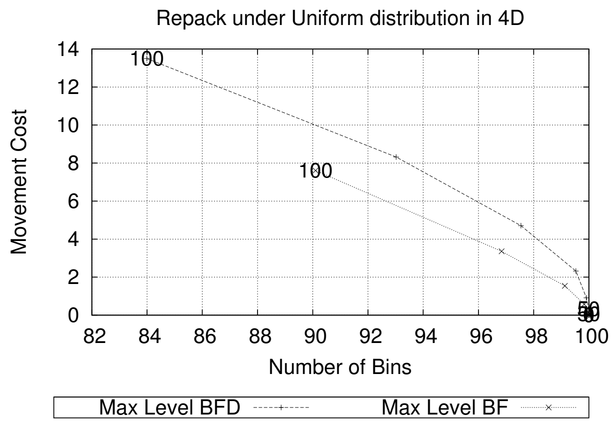
(b)

Fig. 11: *Performance of* Repack *using kMaxLevel BF and kMaxLevel BFD on distributions: (a) Caprara 4, (b) Caprara 5*

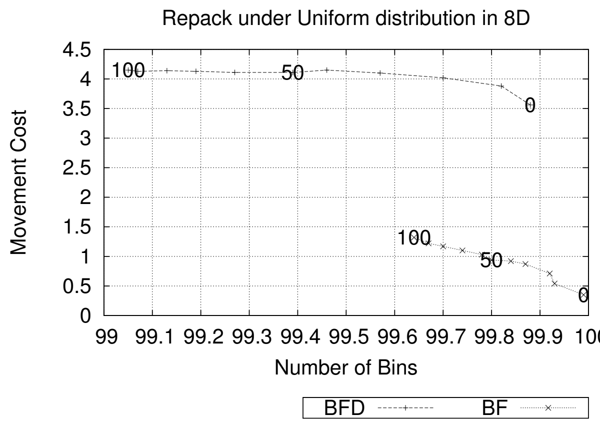
Fig. 12: *Performance of* Repack *using kMaxLevel BF and kMaxLevel BFD on distributions: (a) Caprara 7 (b) Caprara 8*



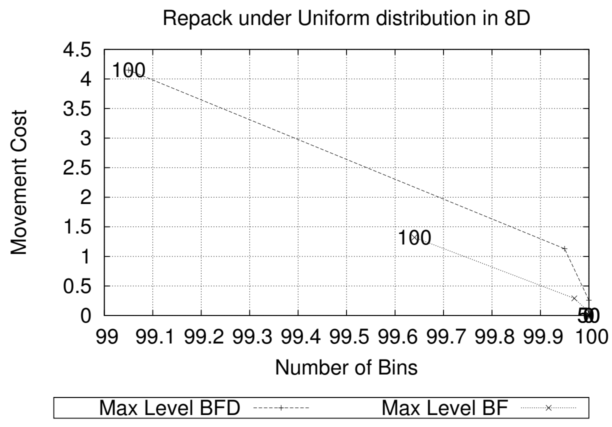
(a)



(a)



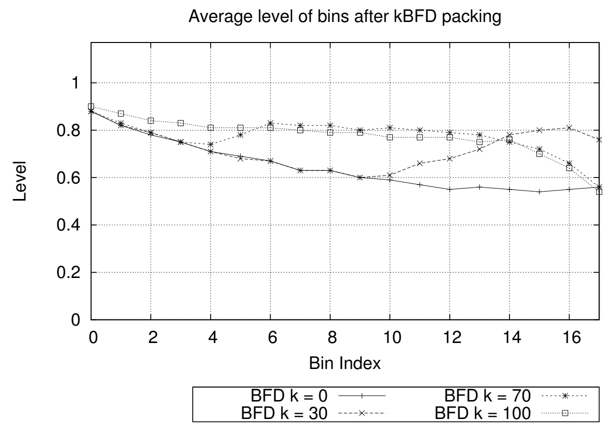
(b)



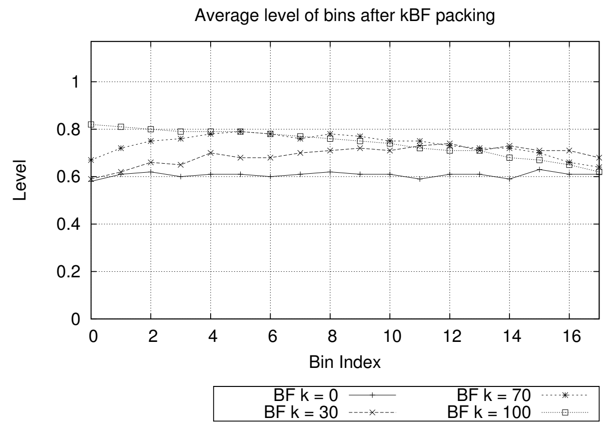
(b)

Fig. 13: *Experiments with* Repack *for 4 and 8 dimensional inputs*

Fig. 14: *Experiments with* Repack *for 4 and 8 dimensional inputs*

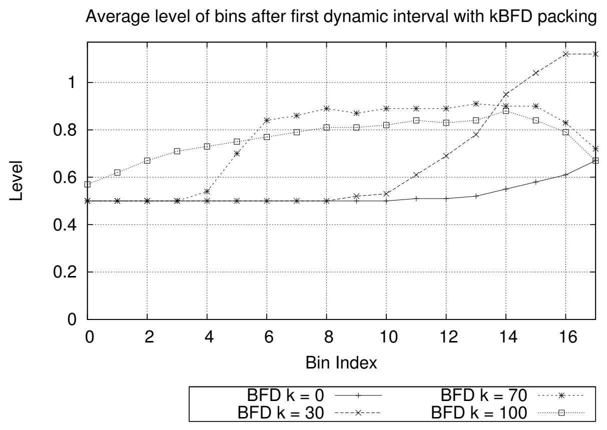


(a)

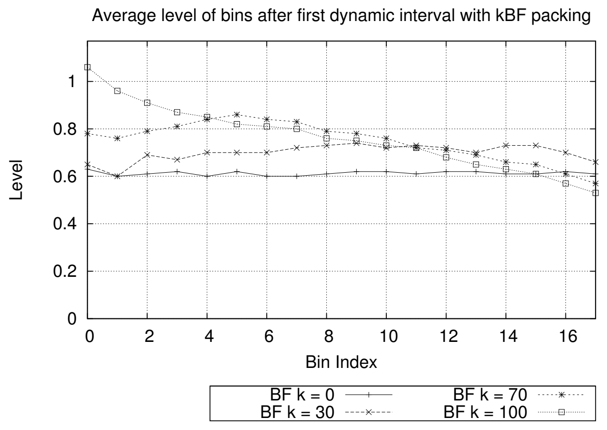


(b)

Fig. 15: *Figures show average levels to which bins are filled after the initial packing with* Repack *(a) kBFD and (b) kBF.*

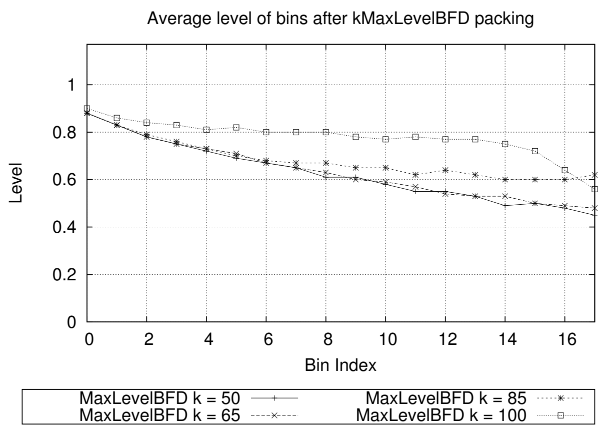


(a)

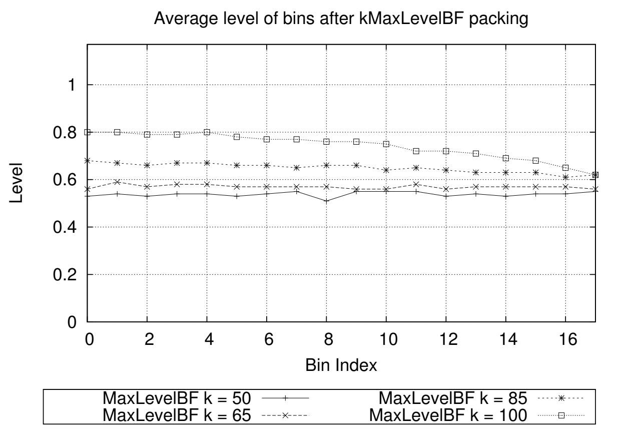


(b)

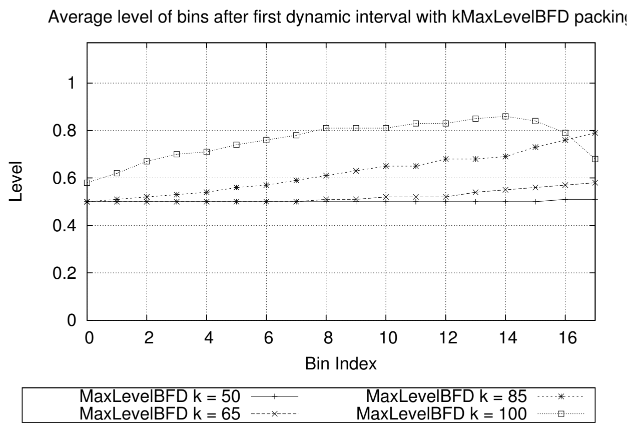
Fig. 16: *Figures show average levels to which bins are filled with* Repack *(a) kBFD and (b) kBF after the first dynamic interval when their profiles are randomly changed. As seen in the graph (a), the level in bins with higher index increases disproportionally compared to bins with lower index (which are opened first).*



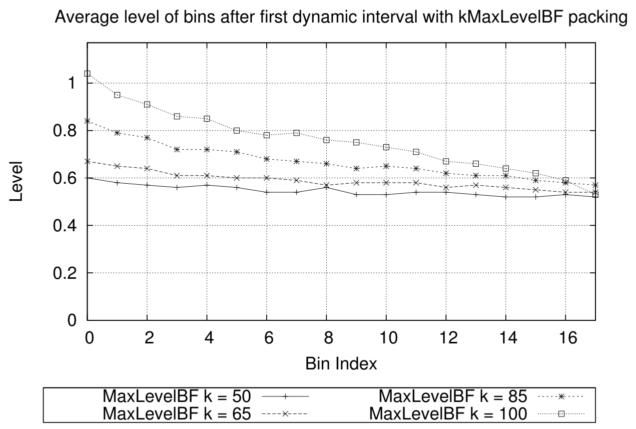
(a)



(b)



(c)



(d)

Fig. 17: *Figures on the top row show average levels to which bins are filled after the initial packing with* Repack *(a) kMaxLevel BFD and (b) kMaxLevel BF. Figures on the bottom row show average levels to which bins are filled with* Repack *(c) kMaxLevel BFD and (d) kMaxLevel BF after the first dynamic interval when their profiles are randomly changed. As seen in the graphs on the bottom row, the levels in most bins change more evenly compared to* $kBFD$