Efficient Bin Packing Algorithms for Resource Provisioning in the Cloud

Shahin Kamali

Massachusetts Institute of Technology, Cambridge, MA 02139, USA
skamali@mit.edu

Abstract. We consider the Infrastructure as a Service (IaaS) model for cloud service providers. This model can be abstracted as a form of online bin packing problem where bins represent physical machines and items represent virtual machines with dynamic load. The input to the problem is a sequence of operations each involving an insertion, deletion or updating the size of an item. The goal is to use live migration to achieve packings with a small number of active bins. Reducing the number of bins is critical for green computing and saving on energy costs. We introduce an algorithm, named HarmonicMix, that supports all operations and moves at most ten items per operation. The algorithm achieves a competitive ratio of 4/3, implying that the number of active bins at any stage of the algorithm is at most 4/3 times more than any offline algorithm that uses infinite migration. This is an improvement over a recent result of Song et al.[12] who introduced an algorithm, named VISBP, with a competitive ratio of 3/2. Our experiments indicate a considerable advantage for HarmonicMix over VISBP with respect to average-case performance. HarmonicMix is simple and runs as fast as classic bin packing algorithms such as Best Fit and First Fit; this makes the algorithm suitable for practical purposes.

1 Introduction

We consider Infrastructure as a Service (IaaS) model in the cloud which has received increasing attention in the past few years. In this model, a cloud service provider such as Amazon EC2 rents virtual machines (VMs) to clients. Each VM is capable of running several applications with dynamic loads that vary by the time. The total load of applications encapsulated in a VM defines the *load* of the VM. The applications are unpredictable in the sense that their load and the pattern of their changes cannot be predicted in advance. In other words, the load of VMs is not known beforehand and changes over time. A service provider has to assign VMs into physical machines (PM's) so that the total load of all VMs in each machine is no more than the uniform capacity of PM's. In other words, servers should not be overloaded in order to avoid bottlenecks in the system and to balance the load between PMs. Moreover, the number of PM's that are used to host VMs is desired to be as small as possible. This objective is important for green computing and reducing energy costs. Particularly, inactive PMs which do not host any VM can hibernate in fractions of a second [10] and hence save on energy costs. The IaaS model, as described above, is closely related to the classic bin packing problem.

In the bin packing problem, the input is a set of *items* each having a *size* in the range (0,1]. The goal is to place these items into a minimum number of bins of uniform capacity. In the IaaS model, items represent VMs; item sizes represent the load of VMs; and bins represents PMs. The bin packing problem requires the total size of items in each bin to be at most equal to the unique capacity of bins, and the objective is to place items into a minimum number of bins. In the online version of the problem, item sizes are not known in advance. Instead, they form a sequence that is revealed item by item. An online algorithm should place an item into a bin without any knowledge about the forthcoming items. In the IaaS model, VMs' loads are not known in advance; hence, the online bin packing is more relevant compared to the offline bin packing where item sizes are known beforehand. An example of an online bin packing algorithm is Next Fit which keeps one *active* bin and places an incoming item in the active bin if it has enough space; otherwise, it closes the bin and opens a new active bin. First Fit is another online algorithm that places each item into the first bin, in the order that they are opened, which has enough space (and opens a new bin if required). Best Fit is similar to First Fit except that it maintains bins in the decreasing order of their *levels*, where the level of a bin is total size of items in it. Harmonic algorithm has a parameter K, where K is a positive integer, and partitions the unique interval into sub-intervals $(1/2, 1], (1/3, 1/2], \ldots, (1/(K+1), 1/K], (0, 1/K]$, and applies a separate Next Fit strategy for items with sizes in each sub-interval.

Competitive analysis is the standard approach for comparing online algorithms. For an online algorithm $\mathbb{A}$, we use $\mathbb{A}(\sigma)$ to denote the number of bins opened by $\mathbb{A}$ for packing a sequence σ. Similarly, we use $\textsc{Opt}(\sigma)$ to denote the number of bins opened by an optimal algorithm $\textsc{Opt}$ for packing σ. In the asymptotic sense, the value of $\textsc{Opt}(\sigma)$ is assumed to be large and the *asymptotic* competitive ratio of $\mathbb{A}$ is defined as the maximum value of $\frac{\mathbb{A}(\sigma)}{\textsc{Opt}(\sigma)}$ for any sequence σ[1] . Next Fit has a competitive ratio of 2, Best Fit and First Fit both have competitive ratio 1.7 [7], and the competitive ratio of Harmonic converges to 1.69 for large values of K [8].

In the standard setting for online bin packing, the decisions of an online algorithm are irrevocable and an item in a bin B cannot be moved to another bin B'. In the IaaS model, however, the dynamic size of VMs requires moving them between servers to avoid overloaded PMs (e.g., when the load all VMs hosted by a PM increase). *Live migration* [3] enables moving VMs between PMs without interrupting applications running inside them. Different strategies are introduced for live migration (see, for example, Sandpiper [13] and VectorDot [11]). However, these approaches are merely focused on load balancing and do not consider green computing. In this paper, we study the online bin packing algorithms for the IaaS model, which is defined as follows.

Definition 1. *In the IaaS model of bin packing, the input is an online sequence of* operations *on items (VMs) with sizes (loads) in the range* $(0, 1]$*. Each operation involves either inserting an item to any bin (PM), removing an item from a given bin, or updating the size of an item from* x *to* y*. Upon applying each operation, an online algorithm can use live migration to move a* constant *number of items between bins. The size of items in each bin at any given time should not be more than the uniform capacity of bins. The goal is to achieve packings with minimum number of bins (active PMs).*

[1]Throughout the paper, by competitive ratio, we mean asymptotic competitive ratio. For results related to the *absolute* competitive ratio of bin packing algorithms, we refer the reader to [4,5].

1.1 Previous Work and Contribution

Gambosi et al. [6] studied a version of online bin packing where only insertion and deletion are allowed. They introduced an online algorithm with a competitive ratio of at most 4/3. Unfortunately, their algorithm is quite complicated and does not seem suitable for practical purposes. Moreover, it does not support update operation. An update operation can be simulated with a delete and then an insert operation. However, as pointed out in [12], this might require moving a large number of items between bins. For example, consider a packing in which there are two bins each having an item of size $0.5 + \epsilon$ and $0.5/\epsilon - 1$ items of size ϵ. Other bins in the packing each include $1/\epsilon - 1$ items of size ϵ. If the size of one of the items of size $0.5 + \epsilon$ increases to $0.5 + 2\epsilon$, its bin gets overloaded. To fix the packing, it suffices to move an item of size ϵ to another bin. However, if we delete and re-insert the updated item, either an extra bins should be opened or at least $0.5/\epsilon$ items should be moved.

The IaaS model of bin packing has been recently studied by Songe et al. [12]. There, the authors introduced an algorithm, called bin packing with variable-sized items (VISBP), which has a competitive ratio of 1.5 and supports all three operation. Although this algorithm uses live migration to improve over the lower bound 1.54037 [1] for competitive ratio of purely online algorithms, it leaves a lot of space for improvement. In particular, we show that live migration can be used more effectively to improve over the competitive ratio and, more importantly, the average-case performance of VISBP.

In this paper, we apply more complicated packing techniques to introduce an algorithm, called HarmonicMix, for the IaaS model. Recall that Harmonic algorithm for bin packing includes i item in the range $(1/(i + 1), 1/i]$ in a bin of type i. This particular structure makes the algorithm suitable for dynamic packing as items of same type can replace each other in their harmonic bin. Unfortunately, Harmonic algorithm has a poor average-case performance [9] when compared to classic algorithms such as Best Fit and First Fit. To address this issue, in HarmonicMix, we make use of live migration to improve the packings of Harmonic algorithms. This ensures a good average-case performance in terms of the number of active bins. At the same time, the algorithm moves a small number of items, at most ten items, per operation. We prove that the competitive ratio of HarmonicMix is 4/3, which is better than 3/2 of VISBP. To compare average-case performance of these algorithms, similar to many related works for average-case study of bin packing algorithms (see [4] for a review), we test the algorithms on randomly-generated input sequences . Our experiments indicate that HarmonicMix has an advantage over VISBP, not only in the worst-case, but also in the average case.

2 HarmonicMix Algorithm

In this section, we introduce and analyse the HarmonicMix algorithm. Similar to the previous works on dynamic bin packing (see, e.g., [12,6]), we assume item sizes are larger than a fixed value. For example, in the VISBP algorithm of [12], this fixed value is defined to be 1/6. Items with sizes at most 1/6 are grouped together to form *multi-items* with sizes in the range $(1/6, 1/3]$. For HarmonicMix, we define this fixed value to be 1/8, i.e., we group items of size at most 1/8 into multi-items with sizes in the range

$(1/8, 1/4]$. This can be done with no computing overhead. In what follows, we always assume item sizes are larger than 1/8. Before introducing the algorithm, we describe a general family of packings called *valid packings*. We prove that any algorithm that maintain a valid packing has a competitive ratio of at most 4/3. Later, we will show how to maintain a valid packing by moving a small number of items after each operation.

2.1 Valid Packings

In what follows, we refer to an item as being *large* if it is larger than 1/2, *medium* if it is in the range $(1/3, 1/2]$, *small* if it is in the range $(1/4, 1/3]$, and *tiny* if it is in the range $(1/8, 1/4]$. We correspond each bin with the largest item in the bin. For example, a bin is medium if it includes a medium item and no large item. A given packing of n items is valid if the following conditions hold. We use the term 'almost all' for a set of bins to indicate all bins in the set except potentially a constant number of them.

1 Almost all medium bins include two medium items and possibly one tiny item.
2 Almost all small bins include three small items.
3 Almost all tiny bins have a level of at least 3/4.
4 For almost all large bins like B that does not contain a medium or small item, and for any medium or small item y in bins other than large bins, we have $x + y > 1$, where x is the size of the large item in B.
5 For almost all large or medium bins like B either $level(B) \geq 3/4$ or there is no tiny bin in the packing.

Intuitively, properties 1-3 can be maintained by placing medium, large, and tiny items in separate bins in a similar fashion as the Harmonic algorithm does. Property 4 implies that a large item should be accompanied with a medium or a small if possible; if it is not possible, then the item might be accompanied by tiny items. Property 5 implies that tiny items should be placed in large and medium bins to ensure a level of at least 3/4 for these bins. In other words, there is a tiny bin in the packing only when the level of all large and medium bins is 3/4 or more.

Lemma 1. *Any algorithm $\mathbb{A}$ that maintains a valid packing has a competitive ratio of at most 4/3.*

Proof. We consider the following two cases for a valid packing P of an input sequence σ and prove the claim for each case separately.

Case I: Assume P includes a tiny bin. We prove that the level of all bins, except possibly a constant number of them, is at least 3/4. This gives a competitive ratio of 4/3 for the packing. Property 2 implies that almost all small bins include three small items, i.e., they have a level in the range (3/4,1]. Property 3 indicates that tiny bins also have level 3/4 or more. Property 5 implies that any large or medium bin B has level 3/4 or more; otherwise, any of the tiny items placed in the tiny bin should have been placed in B.

Case II: Assume P does not include a tiny bin. Consider a fixed optimal packing of σ. In this packing, we refer to the large items that are accompanied by a small or a medium item as *blue* large items and refer to the rest of large items as *red* large items. For each item x, we define a *weight* $w(x)$ for x as follows. For a red large $w(x) = 1$, for a blue large item $w(x) = 5/6$, for a medium item $w(x) = 1/2$, for a small item $w(x) = 1/3$, and for tiny items $w(x) = 0$. Let $W(\sigma)$ denote the total weight of items in σ. We prove $\mathbb{A}(\sigma) \leq W(\sigma) + c$ for some constant c and $\text{OPT}(\sigma) \geq 3/4 \times W(\sigma)$. From these two inequalities, we conclude $\mathbb{A}(\sigma) \leq 4/3 \times W(\sigma) + c$, which completes the proof.

First, we prove $\mathbb{A}(\sigma) \leq W(\sigma) + c$. We show that items in almost all bins in the packing of $\mathbb{A}$ have an average total weight of at least 1. By property 1, almost all medium bins include two medium items. The total weight of these items would be $2 \times 1/2 = 1$. Similarly, by property 3, almost all small bins include three items, each with weight 1/3. The total weight would be $3 \times 1/3 = 1$. There is no tiny bin in Case 2. It remains to show average the weight of items in large bins is at least 1. Let R and B respectively denote the number of red and blue large items; the total contribution of large items to the total weight of all large bins is $R + 5/6 \cdot B$. We claim that at least $B/2$ of large bins include a medium of a small items. If that is true, the contribution of these small/medium items to the weight of large bins would be at least $B/2 \times 1/3 = B/6$. Hence, the total weight of items in large bins would be at least $R + 5/6 \cdot B + B/6 = R + B$. Since the algorithm opens $R + B$ large bins, the average weight of large bins would be at least 1. To prove the claim, we consider set S_l formed by large items and set S_{ms} formed by the union of medium and small items in the input sequence. By definition of blue bins, we have $|S_l| \geq B$ and $|S_{ms}| \geq B$. Let S_l^* and S_{ms}^* respectively denote the smallest $\lfloor B/2 \rfloor$ items of S_l and S_{ms}. For any pair $(x, y), x \in S_l^*, y \in S_{ms}^*$, we have $x + y \leq 1$. To sea that, consider the blue large item z which has median size among the blue large items. There are roughly $B/2$ blue large items smaller than z and $B/2$ small/medium items smaller than $1 - z$. So, all $B/2$ items of S_l^* fit with all $B/2$ items of S_{ms}^*. By property 4, the algorithm tends to place medium/small items in large bins (and for that, they have priority over tiny items). Hence, at least $B/2$ medium/small items are placed in large bins and the claim follows.

Next, we prove $\text{OPT}(\sigma) \geq 3/4W(\sigma)$. We show that any given bin in the fixed optimal packing has a total weight of at most 4/3. By definition, bins with red large items in the optimal packing do not include medium or small items. They might contain tiny items which do not contribute to the total weight. So, the total weight of large bins with a red item is one. Next, consider bins in the optimal packing that includes a blue large item. These bins include at most one other item, i.e., a medium or a small item (tiny items are ignored as their weight is zero). In the former case, the weight of the bin would be $5/6 + 1/2 = 4/3$. In the latter case, the weight would be $5/6 + 1/3 < 4/3$. Next, assume a bin without large items. It might contain 1) two medium items with total weight of one; 2) one medium and two small items with total weight $1/2 + 2/3 < 4/3$; 3) three small items with a total weight of one. In all cases, the total weight of bins is at most 4/3. So, the total weight W of all items is distributed between at last $3/4 \cdot W$ bins. □

2.2 Nice Packings

The HarmonicMix algorithm maintains a certain type of valid packings, called *nice packings*, which we describe here. By property 1 of valid packings, almost all medium bins include two medium items. These two items would have a total size in the range (2/3,1]. In order to fulfill property 5 of a valid packing, the bin might also include a tiny bin. This implies that each medium bin has two *spots* for two medium items and one spot for a tiny item. The tiny spot might be empty but each medium spot includes a medium item. Similarly, property 2 indicates that small bins include three spots for small items, and in almost all small bins the three spots are occupied (i.e., there is no empty spot). Each large bin includes a non-empty spot for a large item. There is also a spot for a medium or a small item. We call this spot medium/small spot (there is no priority between medium and small items for occupying this spot). There are also two tiny spots which are filled with tiny items only if the medium/small spot is empty. This implies that if there is a medium/small item that can fill the medium/small spot, the tiny spots need to be empty.

Note that the above description for bin spots is consistent with the definition of a valid packing. In a nice packing, in addition to the five properties of valid packings, we require that a tiny item be placed in the tiny spot of a large or medium bin, and only if it is not possible then it is placed in a tiny bin. As an example, consider a large bin B which includes an item of size $5/6 - \epsilon$ for some small positive value of ϵ. Clearly, no medium/small item fits in the remaining space of the bin. However, a tiny item x of size $1/6 + \epsilon$ does fit in B. For a valid packing, it is not required to place x in B because property five holds (since B has level more than 3/4). However, to have a nice packing, we require x to be in the tiny spot of B rather than a tiny bin. This property of nice packings is not used in our worst-case analysis; however, it is essential for having a good average-case performance.

In addition to the above-mentioned property, in a nice packing, tiny bins are more structured in the following sense. Recall that an item is tiny if it is in the range $(1/8, 1/4]$. We further partition this interval into sub-intervals $T_1 = (1/5, 1/4]$, $T_2 = (1/6, 1/5]$, $T_3 = (1/7, 1/6]$, and $T_4 = (1/8, 1/7]$. Each tiny bin of a nice packing includes tiny items of the same intervals. We say a tiny bin has type T_i if it includes tiny items of type T_i ($i \in \{1, 2, 3, 4\}$). At any given time, all tiny bins of type T_i include $i + 3$ items of type T_i. The only exception is the most recently opened bin, called the *active bin of type* T_i, which might include less than $i + 3$ items. This way, all tiny bins, except four of them (the active bins), have level $4/5$ or more, which ensures property 3 holds. We extend the notion of active bins to medium and small items. Recall that there are two medium spots in almost all medium bins. The only potential exception is the most recently opened bin, which we call the *active medium bin*. Similarly, there are three small spots in each small bin, again, with the exception of one *active small bin*. Figure 1 provides an illustration of a nice packings.

Any of the insertion/deletion/update operations might result in a packing which is not nice any more. To fix this, we apply live migration to maintain a nice packing. In many cases, this involves moving an item from an active bin to another bin of the same type. This might result in an empty active bin; in this case we declare another bin of the

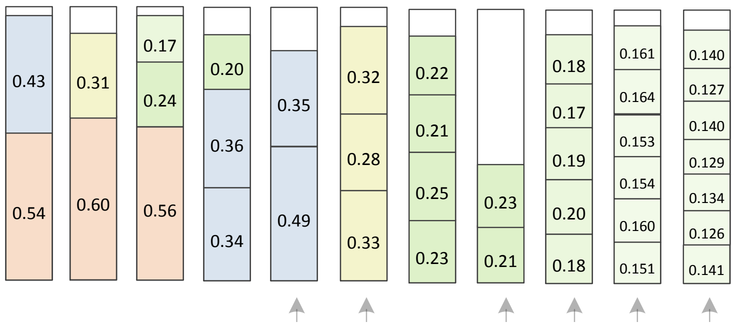


Fig. 1. An example of a nice packing. Colors of items indicate their types (red for large, blue for medium, amber for small, and different shades of green for subfamilies of tiny items). The arrows point to active bins of different groups. Assume we remove the medium item 0.43 from the first bin. To maintain a nice packing, the empty medium/small spot should be filled with a medium item (e.g., 0.35 in the active medium bin) or a small item (e.g., 0.32 in the active small bin).

same type as the new active bin. Similarly, upon inserting an item to the active bin, we might need to open a new bin and declare it as the new active bin.

2.3 Insert/Delete Operations

In what follows, we describe how HarmonicMix updates maintains a nice packing after an insert or a delete operation.

Lemma 2. *It is possible to maintain a nice packing after an insert/delete operation by moving at most five items per operation.*

Proof. To prove the lemma, we discuss how each operation separately.

insert-tiny (no move) Assume we want to insert a new tiny item of size x. To maintain a nice packing, first we check if there is a tiny spot in large or medium bins in which x fits. If there is, we place x there; otherwise, we place x into the active bin of its type. No item is moved from the packing. If x does not fit into the active bin, the level of the bin is more than 3/4 and property 3 holds. One can easily check that the other properties of a valid packing also hold. Note that no item is moved as a result of this operation.

remove-tiny (two moves) Assume we want to remove a tiny item x from a bin B. If B is an active bin, we remove x and no other item is moved. The packing remains valid since the properties of valid packing do not apply to active bins. If B is a non-active tiny bin, we fill the empty spot of x in B with an item x' in the active bin of the same type as B. Since x and x' belong to the same sub-class of tiny items, the packing remains valid. Next, assume x is in the tiny spot of a large or a medium bin. If we can replace x with another item x' located in a tiny bin, we move x's to B. This might require moving another item x'', from the active bin of the same sub-class of x', to the empty spot of x'. In total, at most two items are moved.

insert-small (two moves) To insert a small item x, we first check if there is a large bin B with an empty medium/small spot in which x fits. If there is, we place x in B and remove at most two items from the tiny spots of B and re-insert them; this would require moving at most two items. If there is no such large bin B in which x fits, we simply place x in the active small bin.

remove-small (five moves) If we remove a small item from the active small bin, the packing remains nice and no item is moved. To remove a small item x from a non-active small bin B, we simply replace x with a small item x' in the active small bin. This requires moving one item, i.e., x'.
Next, assume a small item x is removed from a large bin B. We might need to fill the empty spot of B with a medium or a small item in a non-large bin. If a small item is moved, as discussed above, at most one other item is moved, i.e., two items are moved in total. If a medium item is moved, as will be discussed later (see the first paragraph of remove-medium), we might need to move at most four other items to fix the packing, i.e., at most five items are moved in total.

insert-medium (two moves) To insert a medium item x, we first check to see if there is an empty medium/small spots in any of the large bins in which x fits. If there is such spot in a bin B, we place x in B. In case B includes one or two tiny items, we remove them from B and re-insert them to the packing. As suggested above, no item is moved after inserting tiny items. So, at most two items are moved. Next, assume there is no spot for x in the large bins. We place x in the active medium bin and open a new bin if required. In case we open a new bin, the previous active bin will have an empty spot which might be filled with an item in a tiny bin. Removing an reinserting such tiny item requires at most two moves.

remove-medium (five moves) Assume a medium item x is removed from a medium bin B. If B is the active medium bin, no item needs to be moved. If B is a non-active bin, the empty spot of x in B is filled with a medium item x' from the active bin. This might result in an overloaded bin when $x' > x$; this happens only if there is a tiny item z in B. To fix the packing, we remove z from B and re-insert it; this only requires moving z since no item is moved after re-inserting a tiny item. To ensure a nice packing, we might need to fill the empty spot of z with another tiny item z' in a tiny bin B'. This requires filling the empty spot of z' in B' with another tiny item z'' in the active tiny bin of the same sub-class. In total, we moved at most four items (x', z, z', and z'').
Next, assume a medium item x is removed from a large bin B. To maintain a valid packing, we might need to fill the empty spot of B with a medium or a small item from the non-large bins. If a medium item is moved, as discussed in the above paragraph, we might need to move at most four other items to fix the packing, i.e., at most five items in total. If a small item is moved, at most one other item is moved, i.e., two items are moved in total.

insert-large (five moves) Assume we want to insert a large item x. We open a new bin B for x; this bin would have an empty medium/small spot. If there is a medium/small item y in non-large bins which fits in the empty spot, we move y to B. As discussed

earlier, removing a medium or small item from a non-large bin requires moving at most four other items, i.e, we move at most five items in total.
If there is no medium or small item that fits in the medium/small spot, we move tiny items into the two tiny spots of B. This requires moving at most two tiny items from tiny bins, and each potentially need moving another tiny item from the active bin of the same sub-class to the the new empty spots. In total, we move at most four items.

remove-large (three moves) Assume we remove a large item x from a large bin B. To maintain a nice packing, we remove other items in B and re-insert them to the packing. Assume the medium/small spot in B is occupied with an item y. Re-inserting y to the packing requires moving at most two other items. In total, at most three items are moved. If the medium/small spot in B is empty, there are at most two items in the tiny spots of B. Removing and re-inserting these items requires at most two moves. □

2.4 Update Operations

A simple approach to implement updates is to remove the updated item and re-insert it. By Lemma 2, each operation requires moving at most five items. So, this approach requires moving at most ten items per update. While in the worst-case we might indeed need ten moves, for most cases, we can maintain a nice packing with less overhead.

Lemma 3. *Updating the size of an item so that it becomes or remains a tiny item requires at most five moves. If an item becomes or remains small or medium, at most seven moves are required. If a tiny items becomes large, at most seven moves are required. If a medium or small item becomes large, at most ten moves are required. If a large item remains large after an update, at most six moves are required.*

Proof. As before, we discuss each operation separately.

update-to-tiny (five moves) Assume we update the size of an item x so that it becomes tiny after the update. We remove x from the packing (at most five moves) and re-insert it (no move). In total, we move at most five items.

update-to-small and update-to-medium (seven moves) Assume we update the size of an item x so that it becomes small or medium after the update. We remove x from the packing (at most five moves) and re-insert it (at most two moves). In total, we move at most seven items.

update-to-large (ten moves) Assume we update the size of an item x so that it becomes large after the update. If x is tiny before the update, we remove x, using at most two moves, and re-insert it, using at most five moves. The number of moves will be at most seven.
Assume x was a small item in a small bin B before the update. After the update, we remove x (the first move), and fill its empty spot with an item from the active small bin (the second move), and then re-insert x as a large item (at most five additional moves). In total, at most seven moves are required. If x was a small item in a large bin before the update, we remove and re-insert it with at most ten moves.

Assume x was a medium item in a medium bin B before the update. So, x was placed with another medium item x' in B. If we have $x + x' \leq 1$ after the update, we maintain the nice packing by removing and re-inserting the tiny item of B (if there is one). That would require moving one item. If $x + x' > 1$ after the update, we need to remove x' from B. If there is another medium item x'' in a medium bin B'' so that $x + x'' \leq 1$, we swap x and x' (the first two moves). Moreover, if there was an item in the tiny spot of B, we remove and re-insert that item (the third move). Finally, if B'' is overloaded after the insertion, we remove and re-insert the tiny items in it (the fourth move). In that case, the tiny spot of B'' might be filled with another tiny item in a tiny bin (the fifth move). The resulting empty tiny spot in the tiny bin might be filled with an item in the active bin of the same sub-class (the sixth move).
Assume x was a medium item in a large bin B before the update. We simply remove x from B (at most five moves) and re-insert it as a large item (at most five moves). In fact, there are instances in which, to maintain a nice packing, these ten moves are required to maintain a nice packing.
Assume x was large before the update and its size is increased. If the medium/small spot is empty, we might need to move and replace at most two tiny items from the bin using at most six moves. Next, assume the medium/small spot is filled with an item y. After the increase, if the bin B is not overloaded, the packing is still nice. Otherwise, we need to remove y from B. Assume there is another medium item y', with size smaller than y, that can replace y in B. Swapping y and y' requires two moves. After that, the bin B' of y' might become overloaded. In that case, we remove the tiny item in B', denoted by z', and re-insert it to the packing (the third move). To fill the empty spot of z' in B', we might need to move two more tiny items (the fourth and the fifth moves). In total, we move at most five items. If there is no item y' to swap with y, we remove and re-insert y using two moves. Moreover, at most two tiny items will be moved to B, each requiring moving at most one other item from the active bin of their sub-class. In total, we move at most six items.
Next, assume x was large before the update and its size is decreased. If the medium/small spot was occupied before the update, the packing remains nice (no move). Otherwise, we should check if there is a medium or small item y in non-large bins that can fill the medium/small spot. If there is, we move y to B (the first move). The empty spot of y in its previous bin B' will be replaced by another item y' from the active medium/small bin (the second move). If y is medium, this might cause an overflow in B' which can be fixed by moving the tiny item z of B to the active bin of the same type (the third move) and replacing it with another tiny item z' (the fourth move). There might be tiny items in the tiny spots of B before the update. We have to remove these items from B and re-insert them to the packing (the fifth and the sixth moves). Note that inserting a tiny item does not cause extra moves. If there is no medium/small item y to be placed in B after the update, the tiny spots in B are filled with items in tiny bins, using at most four moves. In total, we move at most six items when the size of a large item decreases. □

We use the case analysis presented in Lemmas 2 and 3 to devise the HarmonicMix algorithm. From Lemmas 1,2, and 3, we conclude the following theorem:

Theorem 1. *HarmonicMix has a competitive ratio of at most 4/3 for the IaaS model of bin packing. The number of items moved for each operation is at most ten.*

3 Experiments

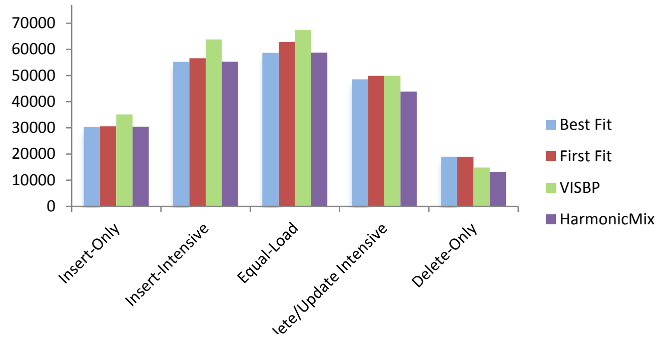
The results in the previous section imply that, in the worst case, HarmonicMix has an advantage over VISBP, i.e., it opens a smaller number of bins. In this section, we experimentally compare the two algorithms to study their average-case performance. We generate random sequence of operations which involve items whose sizes are also generated independently at random. The size of items is in defined to be in the range $(1/6, 1]$ so that HarmonicMix and VISBP work on the same set of items (otherwise, multi-items of the two algorithms will be different).

We present three experiments. In the first experiment, item sizes are generated uniformly at random from the range (1/6,1]. In the second experiment, item sizes follow a normal distribution with mean 0.5 and standard deviation 1. In the third experiments, item sizes take values from the set $\{1/1000, 2/1000, \ldots, 1\}$ following a Zipfian distribution with skew parameter $s = 1.1$ (the smaller the size, the more frequent the item). Each experiment has five phases. Each phase includes $n = 50,000$ operations. In phase one, n random numbers are inserted to the packing. No item is removed or updated during this *insert-only phase*. In the second phase, with a chance of 90 percent an insert operation is performed while chances of delete and update are equal to 5 percent. We call this the *insert-intensive phase*. In the third phase, with a chance of 50 percent, an insert operation is applied and with a chance of 50 percent, a delete or an update operation is applied (each with the same chance of 25 percent). We call this phase, *equal-load phase*. In the fourth phase, called *delete/update intensive phase*, with a chance of 10 percent an insert is applied and with a chance of 90 percent, a delete or update operation is applied (each with the same chance of 45 percent). Finally, in the last phase, called *delete-only phase*, only delete operations are applied.

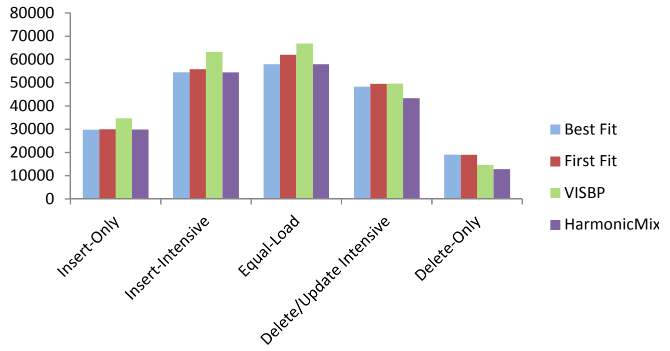
Next, we describe how these insert/delete/update operations are defined. On an insertion, a new item of random size is inserted to the packing. On a deletion, an existing item in the packing is selected uniformly at random and removed. On an update, an existing item is selected randomly and and a random value in the range $(-0.1, +0.1)$ is added to the size of x. If the size of x becomes larger than 1 or smaller than 1/6, another random change is added to have the size in the desired range.

Besides VISBP and HarmonicMix, we also modified Best Fit and First Fit to include them in our experiments. In contrast to the classic algorithms, our modified algorithms use live migration when the size of an item is updated (increased) so that the bin is overloaded, i.e., its level is more than 1. Only in this case, the item with the updated size is removed and re-inserted in the packing. Moreover, when the level of a bin becomes zero (when all items in the bin are removed) the bin is removed from the packing, i.e., it is not counted as one of the bins used by the algorithms.

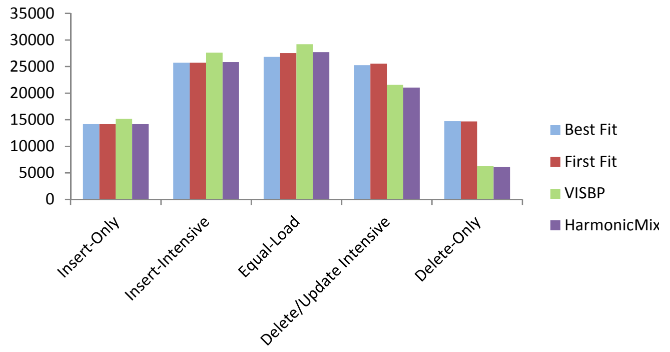
Figures 2 show the number of bins for each algorithm at the end of each phase. One interesting observation is that the packings of Best Fit and First Fit are better than VISBP at the initial phases. This is partially because VISBP is designed in a way to improve the worst-case performance with a minimum number of live migrations. Moreover, Best Fit and First Fit algorithm are known to be optimal algorithms for packing sequences that are generated randomly (see, e.g., [2]). In particular, when items are removed and new random items, the resulting empty spots in the packings of Best Fit and First Fit is likely to be filled with new items (which have the same distribution on their



(a) Experiment 1 (Uniform item sizes)



(b) Experiment 2 (Normal item sizes)



(c) Experiment 3 (Zipfian item sizes)

Fig. 2. The average number of active bins at the end of each phase of the experiments.

sizes). In the last two phases in which delete operations are more frequent than inserts, VISBP shows its advantage over Best Fit and First Fit. HarmonicMix has advantage over other algorithms in almost all phases. The algorithm has a visible advantage over VISBP, and its advantage over Best Fit and First Fit is evident in the last two phases.

We also counted the total number of times that items have been moved in VISBP and HarmonicMix. The average number of migrations per operation in experiment 1 is 0.234 for VISBP and 0.457 for HarmonicMix. Similar numbers are observed for experiments 2 and 3. One can conclude that HarmonicMix tends to move more items to improve the quality of its packing while VISBP tends to minimize the number of migrations instead. Note that, although VISBP/HarmonicMix move at most seven/ten items per operation in the worst case, the expected number of moves is much smaller.

4 Concluding Remarks

HarmonicMix maintains valid packings that are also nice packings. The algorithm can be modified to maintain valid packings, which are not nice, while moving at most five items per operation. Such algorithm has the same competitive ratio of 4/3 of HarmonicMix. So, in the worst case, the algorithm has an advantage over VISBP and HarmonicMix. However, the new algorithm performs poorly on average. We leave further analysis of this algorithm and other variants of HarmonicMix as a future work.

References

1. Balogh, J., Békési, J., Galambos, G.: New lower bounds for certain classes of bin packing algorithms. Theoret. Comput. Sci. 440–441, 1–13 (2012)
2. Bentley, J.L., Johnson, D.S., Leighton, F.T., McGeoch, C.C., McGeoch, L.A.: Some unexpected expected behavior results for bin packing. In: Proc. 16th Symp. on Theory of Computing (STOC). pp. 279–288 (1984)
3. Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live migration of virtual machines. In: 2nd Symposium on Networked Systems Design and Implementation (NSDI) (2005)
4. Coffman, E.G., Garey, M.R., Johnson, D.S.: Approximation algorithms for bin packing: A survey. In: Approximation algorithms for NP-hard Problems. PWS Publishing Co. (1997)
5. Coffman Jr., E.G., Csirik, J., Galambos, G., Martello, S., Vigo, D.: Bin packing approximation algorithms: survey and classification. In: Pardalos, P.M., Du, D.Z., Graham, R.L. (eds.) Handbook of Combinatorial Optimization, pp. 455–531. Springer (2013)
6. Gambosi, G., Postiglione, A., Talamo, M.: Algorithms for the relaxed online bin-packing model. SIAM J. Comput. 30(5), 1532–1551 (2000)
7. Johnson, D.S.: Near-optimal bin packing algorithms. Ph.D. thesis, MIT (1973)
8. Lee, C.C., Lee, D.T.: A simple online bin packing algorithm. J. ACM 32, 562–572 (1985)
9. Lee, C.C., Lee, D.T.: Robust online bin packing algorithms. Tech. Rep. 83-03-FC-02, Department of Electrical Engineering and Computer Science, Northwestern University (1987)
10. Meisner, D., Gold, B.T., Wenisch, T.F.: Powernap: eliminating server idle power. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 205–216 (2009)
11. Singh, A., Korupolu, M.R., Mohapatra, D.: Server-storage virtualization: integration and load balancing in data centers. In: Proceedings of the ACM/IEEE Conference on High Performance Computing. p. 53 (2008)
12. Song, W., Xiao, Z., Chen, Q., Luo, H.: Adaptive resource provisioning for the cloud using online bin packing. IEEE Trans. Computers 63(11), 2647–2660 (2014)
13. Wood, T., Shenoy, P.J., Venkataramani, A., Yousif, M.S.: Sandpiper: Black-box and gray-box resource management for virtual machines. Computer Networks 53(17), 2923–2938 (2009)