

Online Tree Node Assignment with Resource Augmentation

Joseph Wun-Tat Chan* Francis Y. L. Chin[†] Hing-Fung Ting[†] Yong Zhang[†]

*Department of Computer Science, King's College London, Strand, London WC2R 2LS, UK
joseph.chan@kcl.ac.uk

[†]Department of Computer Science, The University of Hong Kong, Pokfulam Road, Hong Kong
{chin, hfting, yzhang}@cs.hku.hk

Abstract

Given a complete binary tree of height h , the *online tree node assignment problem* is to serve a sequence of assignment/release requests, where an *assignment request*, with an integer parameter $0 \leq i \leq h$, is served by assigning a (tree) node of level (or height) i and a *release request* is served by releasing a specified assigned node. The node assignments have to guarantee that no node is assigned to two assignment requests unreleased, and every leaf-to-root of the tree contains at most one assigned node. With assigned node reassignments allowed, the target of the problem is to minimize the number of assignment/reassignment, i.e., the cost, to serve the whole sequence of requests. This online tree node assignment problem is fundamental to many applications, including OVFS code assignment in WCDMA networks, buddy memory allocation and hypercube subcube allocation.

Most of the previous results focus on how to achieve good performance when the same amount of resource is given to both the online and the optimal offline algorithms, i.e., one tree. In this paper, we consider resource augmentation, where the online algorithm is allowed to use more trees than the optimal offline algorithm. By using different approaches, we give (1) a 1-competitive online algorithm, which uses $(h+1)/2$ trees, and is optimal because $(h+1)/2$ trees are required by any online algorithm to match the cost of the optimal offline algorithm with one tree. (2) a 2-competitive algorithm with $3h/8+3$ trees. (3) an amortized $8/3$ -competitive algorithm with $11/4$ trees. (4) a general amortized $(4/3+\alpha)$ -competitive algorithm with $(11/4+4/(3\alpha))$ trees, for any α where $0 < \alpha \leq 4/3$.

1 Introduction

The tree node assignment problem is defined as follows. Given a complete binary tree of height h , the target is to serve a sequence of requests. Every request is classified as either an *assignment request* or a *release request*. To serve an assignment request, which is associated with an integer parameter $0 \leq i \leq h$, we have to assign it a (tree) node at level (or height) i . To serve a release request, we just need to mark the assigned node free. There are two constraints for the node assignments that is (1) any node can be assigned to at most one unreleased assignment request, (without ambiguity, all assigned requests are assumed unreleased), and (2) there is at most one assigned node in every leaf-to-root path. Fig. 1 gives a valid free node assignment.

The tree node assignment problem can be considered as a general resource allocation problem, which can model the specific problems, such as the Orthogonal Variable Spreading Factor (OVFS) code assignment problem [2, 3, 7, 12, 13, 14, 15, 17], the buddy memory allocation problem [1, 5, 10, 11], and the hypercube subcube allocation problem [6]. The main difference between these problems is how the resource, the nodes at level i , for $0 \leq i \leq h$, are interpreted. In the OVFS code assignment problem, the resource consists of codes of frequency bandwidth 2^i ; in the buddy memory allocation problem,

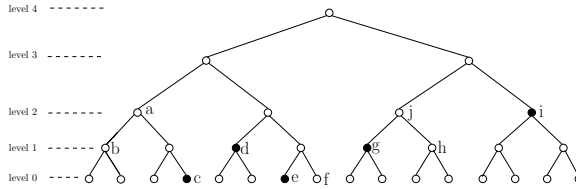


Figure 1: Example of a valid nodes assignment. Filled circles represent assigned nodes.

the resource consists of memory blocks of size 2^i ; in the hypercube subcube allocation problem, the resource consists of subcubes of 2^i processors.

Similar to the memory allocation problem, algorithms for the tree node assignment problem also face the *fragmentation problem*. For example, in Fig. 1, there is no node of level 2 that can be assigned (without violating constraint (2) of node assignments). In fact, we can “defragment” the tree by reassigning the assigned node c to free node f . Then, node a is free to assign to assignment request of level 2. In this paper, we consider the tree node assignment problem where *reassignment* of nodes is allowed. In addition, we design algorithms that serve *all* requests in the request sequence, and we assume that all requests in the request sequence can be served by some algorithm using only one tree of height h .

The performance of an algorithm for the tree node assignment problem is measured by the number assignments/reassignments, which is called the *cost*, carried out by the algorithm. The release operations take no cost, as in the applications, the operation to release a resource is usually negligible when compared with the overhead of assigning/reassigning a resource.

Both the offline and online version of the tree node assignment problem are well studied, especially in the context of OVSF code assignment. In the offline version, the sequence of requests is known in advance, whereas in the online version, the algorithm must process each request without any information about future requests. The offline version of this problem is proved to be NP-hard [16].

Most of the previous work studied the online version of the problem, where performance is given in terms of competitive ratios, i.e., the worst case ratio of the costs between the online algorithm and the optimal offline algorithm. Erlebach et al [7] gave an $O(h)$ -competitive algorithm, where h is the height of the tree, and proved a general lower bound on the competitive ratio of at least 1.5. Forisek et al [8] gave a constant-competitive algorithm, but without deriving the exact value of the constant. Chin, Ting and Zhang [2] gave a 10-competitive algorithm and, in addition, their algorithm guarantees that each request is served with at most 5 assignments/reassignments. They then improved the upper bound by proposing a 6-competitive algorithm [3]. They improved the lower bound of the competitive ratio to $5/3 \approx 1.67$ [2]. Very recently, Miyazaki and Okamoto [14] gave a 7-competitive algorithm and improved the lower bound of the competitive ratio to 2.

In this paper, we focus on the online version of the problem with resource augmentation [9], which means that the online algorithm is allowed to use more trees than the optimal offline algorithm. We assume that the optimal offline algorithm uses one tree, while the online algorithm can use k trees, where $k \geq 1$. The competitive ratio is defined to be the worst case ratio of the cost between the online algorithm with k trees and the optimal offline algorithm with one tree. This problem has been studied before. Erlebach et al [7] gave a 4-competitive algorithm with two trees, and Chin, Zhang and Zhu [4] gave a 5-competitive algorithm with $9/8$ trees.

The main contribution of this paper is to show how the competitive ratio can be further reduced by making use of more trees. In other words, how the future information (offline) can be compensated by extra resources (trees). First, we give an online algorithm with $(h + 1)/2$ trees that matches the cost of the optimal offline algorithm with one tree. In fact, this algorithm even matches the cost of each request with that of the optimal offline algorithm with one tree, as it does not require any reassignments. We further show that for any online algorithm to match the cost of the optimal offline

algorithm with one tree, $(h + 1)/2$ trees are necessary. That implies that our 1-competitive algorithm is optimal in terms of the number of trees used. Then, by using one extra reassignment for each release request to reduce the fragmentation of the assigned nodes in the tree, we can use less number of trees to satisfy the sequence of requests. For particular, we give a 2-competitive online algorithm with $3h/8 + 3$ trees. This algorithm bounds the cost of each request to one, i.e., each assignment request takes one assignment and each release request takes at most one reassignment. These two algorithms with bounded costs for each request are presented in Section 2.

When it is not necessary to bound the cost of individual requests to a constant, we give an amortized $8/3$ -competitive algorithm with $11/4$ trees. This algorithm can be modified to achieve a tradeoff between the competitive ratio and the number of trees used. The result is an amortized $(4/3 + \alpha)$ -competitive algorithm with $(11/4 + 4/(3\alpha))$ trees, for any α where $0 < \alpha \leq 4/3$. These two algorithms are presented in Section 3.

2 Algorithms with bounded cost per request

We present two algorithms in this section. For any request sequence σ where the optimal algorithm can satisfy the requests with one tree,

- the first algorithm uses at most $(h + 1)/2$ trees and incurs a cost of one for each assignment request and a cost of zero for each release request;
- the second algorithm uses at most $3h/8 + 2$ trees and incurs a cost of at most one for each assignment and release request.

Since any algorithm needs to assign a node for each assignment request, the first algorithm is optimal in terms of cost, i.e., 1-competitive. We further show that it is necessary for any online algorithm to use $(h + 1)/2$ trees in order to match the cost of the optimal algorithm with one tree. Since the number of release requests is at most the number of assignment requests, the total cost incurred by the second algorithm is at most twice that of the optimal algorithm with one tree, i.e., 2-competitive.

2.1 Preliminary

We introduce some definitions in this section, which are used in subsequent sections. A node v is called a *free node* if there is no assigned node in any leaf-to-root path going through the node v . A node v is *blocked* if there is a path from the root to a leaf through the node v that contains an assigned node. Node v is also called a *blocked node*, moreover, we say node v is *blocked by an assigned node at some level*. A node at level i or an assignment request that asks for a node at level i is said to have a *bandwidth* of 2^i . It is clear that to serve a set of assignment requests or to accommodate a set of assigned nodes in a tree of height h , the total bandwidth of the requests or nodes has to be no more than 2^h .

2.2 Optimal 1-competitive algorithm

The idea to achieve the optimal cost is to dedicate some subtrees to serve assignment requests of some particular levels. To describe this assignment scheme, we define a *half-tree* to be the subtree rooted at either the left or right child of a root. This online algorithm uses $h + 1$ half-trees. We label the $h + 1$ half-trees from 0 to h . When there is a level- i assignment request with $i < h$, we pick from half-trees 0 to $i + 1$ any free node at level i and assign it to the request. If $i = h$, we assign the root of any tree to the request, as there should be no other assigned nodes. For any release request, we just release the assigned node and mark it free.

The correctness of the algorithm depends on whether, for each level- i assignment request, we can always find a level- i free node from the half-trees 0 to $i + 1$. The following lemma makes sure that it can be done.

Lemma 1. *If the total bandwidth of the assigned nodes is less than 2^h , there is always a free node at level i in the half-trees 0 to $i + 1$ for $i < h$.*

Proof. We prove this lemma by contradiction. Suppose there is no free node at level i in the half-trees 0 to $i + 1$. We prove that the total bandwidth of the assigned nodes is at least 2^h , which contradicts to the assumption.

By the algorithm, half-tree k has assigned nodes at level at least $k - 1$, for $0 < k \leq h$. For half-tree 0, there may be assigned nodes at any level, except the root. If there is no free node at level i in half-tree $i + 1$, then all level- i nodes in half-tree $i + 1$ must be either an assigned node or blocked by an assigned node at level higher than i . Thus, the total bandwidth of the assigned nodes in half-tree $i + 1$ is 2^{h-1} , i.e., the maximum possible bandwidth from a half-tree. Consider half-tree i . The assigned nodes must be at level at least $i - 1$. If there is no free node at level i , at least half of all level- $(i - 1)$ nodes in half-tree i must be either an assigned node or blocked by an assigned node at level higher than $i - 1$. Thus, the total bandwidth of the assigned nodes in half-tree i is at least 2^{h-2} , i.e., half of the maximum possible bandwidth from a half-tree. Using a similar argument, we can show that the total bandwidth of the assigned nodes in half-tree $0 < k \leq i + 1$ is at least $2^{h+k-i-2}$. Note that for half-tree 0, the condition of selecting free nodes for serving assignment requests is the same as that of half-tree 1. The bound of the total bandwidth of the assigned nodes in half-tree 0 is the same as that of the half-tree 1, i.e., it is at least 2^{h-i-1} . As a result, the total bandwidth of the assigned nodes in all half-trees is at least $\sum_{k=1}^{i+1} 2^{h+k-i-2} + 2^{h-i-1} = 2^h$. \square

Theorem 1. *For the online tree node assignment problem, we have an 1-competitive algorithm using $(h + 1)/2$ trees, where the cost of serving each assignment request is one and the cost of serving each release request is zero.*

2.2.1 Lower bound of number of trees to achieve 1-competitiveness

We give an adversary such that the optimal algorithm with one tree serves each assignment request with only one assignment and each release request with no reassignment. At the same time, for any online algorithm that wants to limit the cost as the optimal algorithm, the adversary forces it to use at least $(h + 1)/2$ trees.

The main idea of the adversary is to send assignment requests in ascending order of their levels. The adversary then releases some requests but makes sure that the remaining assigned nodes of low level block a significant part of the trees. Thus, the assignment requests of high level need to be served with extra trees. The adversary is divided into h steps, where in Step i , assignment requests of level i are sent, and then some release requests of level $j \leq i$ follow, except for Step $h - 1$. Over all time, the total bandwidth of the assigned nodes is kept at most 2^h . The details of the adversary is given as follows.

Step 0: The adversary sends 2^h level-0 assignment requests. Any online algorithm must assign 2^h level-0 free nodes. The adversary then releases 2^{h-1} of the level-0 assigned nodes such that $2^{h-1} = 2 \cdot 2^{h-2}$ level-1 nodes are blocked.

Step 1: The adversary sends 2^{h-2} level-1 assignment requests. After the online algorithm has assigned 2^{h-2} level-1 free nodes, the adversary releases 2^{h-2} level-0 and 2^{h-3} level-1 assigned nodes, i.e., half of the assigned nodes at each level with assigned nodes. The release requests make sure that it results in $3 \cdot 2^{h-3}$ nodes blocked at level-2.

...

Step i , for $2 \leq i \leq h - 2$: The adversary sends 2^{h-i-1} level- i assignment requests. After the online algorithm has assigned 2^{h-i-1} level- i free nodes, the adversary releases 2^{h-i-1} level-0 assigned nodes and 2^{h-i-2} level- j assigned nodes for $1 \leq j \leq i$, i.e., half of the assigned nodes at each level with assigned nodes. The release requests make sure that it results in $(i + 2) \cdot 2^{h-i-2}$ nodes blocked at level- $(i + 1)$.

...

Step $h - 1$: The adversary sends one level- $(h - 1)$ assignment requests. (Now, there are $h + 1$ nodes blocked at level $h - 1$.)

It is easy to construct an offline algorithm with one tree so that it serves for the adversary each assignment request with one assignment and each release request with no reassignment.

We prove that for any online algorithm at the end of Step $h - 1$, it has used at least $(h + 1)/2$ trees. It is based on a property (Lemma 2) that at the end of Step i , for $0 \leq i \leq h - 2$, the number of level- $(i + 1)$ node blocked is $(i + 2) \cdot 2^{h-i-2}$. Hence, together with the assigned node resulted in Step $(h - 1)$, there are $(h + 1)$ level- $(h - 1)$ nodes blocked. If the algorithm is maintaining $(h + 1)$ level- $(h - 1)$ nodes, it is using at least $(h + 1)/2$ trees. Therefore, the lower bound of the number of trees necessary for an online algorithm to match the cost of the optimal algorithm is established (Theorem 2).

Lemma 2. *For any online algorithm, at the end of Step i , the number of level- $(i + 1)$ nodes blocked is $(i + 2) \cdot 2^{h-i-2}$, for $0 \leq i \leq h - 2$. (The proof of this lemma is in Appendix.)*

Theorem 2. *No online algorithm can match the cost of the optimal offline algorithm with one tree by using less than $(h + 1)/2$ trees.*

2.3 2-competitive algorithm with bound cost per request

This section gives an online algorithm that uses fewer trees, i.e., $3h/8 + 2$, but comes with a slightly higher competitive ratio, i.e., 2. The main idea of the algorithm is to apply an extra reassignment for any release request to reduce fragmentation of the tree (to reduce blocking bandwidth) by pairing two assigned nodes with unassigned siblings so as to use fewer trees. Precisely, the algorithm serves each assignment and release request with at most one assignment or reassignment. As the number of release requests is at most the number of assignment requests, the total cost of the algorithm is at most twice that of the optimal algorithm.

We design an assignment scheme for the $3h/8 + 2$ trees available to the online algorithm. First, we define an *eighth-tree* to be a subtree rooted at a level- $(h - 3)$ node. All the $3h$ eighth-trees in the $3h/8$ trees are labeled. Six of them are labeled 0 and three of them are labeled i , for $1 \leq i \leq h - 2$. Denote the other two trees as T and T^* . In general, the $3h$ eighth-trees are to handle the assignment requests at level- i for $0 \leq i \leq h - 3$, T for level- $(h - 2)$ and $-(h - 1)$, and T^* for all levels. In particular, at any time, we allow at most one assigned node in each level $0 \leq i \leq h$ of T^* . It enables us to find a free node at level- i of T^* , whenever there is no assigned node at level- i .

The details of the assignment scheme are given as follows.

Assignment request R of level i :

If there is no level- i assigned node in T^* , assign R a level- i free node in T^* .

Otherwise,

- If $i = h - 2$ or $i = h - 1$, assign R any level- i free node in T .
- If $0 \leq i \leq h - 3$, assign R any level- i free node from any eighth-tree labeled k for $0 \leq k \leq i$. If no free node is available, consider the eighth-trees labeled $i + 1$. If there is a level- i free

node v where v 's sibling is an assigned (level- i) node, assign R the free node v . Otherwise, assign R any level- i free node in any eighth-tree labeled $i + 1$. (Lemma 4 shows that an level- i free node always exists in one of the eight-tree with label from 0 to $i + 1$).

Release request R of level i :

Release the node assigned to R and mark it free.

If $0 \leq i \leq h - 3$, consider the following situations for reassignment.

- If there is no assigned node at level i of T^* and there is a level- i assigned node v in an eighth-tree labeled $i + 1$ where v 's sibling is a free (level- i) node, reassign v to a level- i free node of T^* .
- If there are two level- i assigned node u and v in an eighth-trees labeled $i + 1$ where both u 's and v 's siblings are a free (level- i) node, reassign u to v 's sibling.

To ensure the correctness of the algorithm, we show that the followings are true.

1. When T^* contains an assigned node at level i , for $i = h - 1$ or $i = h - 2$, there is always a level- i free node in T . For this case, it is clear as otherwise, the total bandwidth of the assigned nodes is at least 2^h .
2. When T^* contains an assigned node at level i for $0 \leq i \leq h - 3$, there is always a level- i free node in some eighth-tree labeled k , for $0 \leq k \leq i + 1$. This property is proved in Lemma 4

In order to ensure that the Property (2) is true (by Lemma 4), we may spend an extra reassignment after a release request to tidy up the configuration of the assigned nodes in the eighth-trees. We want to maintain a configuration of the assigned nodes in the eighth-trees as in the following lemma.

Lemma 3. *Let $0 \leq i \leq h - 3$. (1) When T^* contains no assigned node at level i , there is no assigned node v at level i of any eight-tree labeled $i + 1$ where v 's sibling is a free node. (2) When T^* contains an assigned node at level i , there is at most one assigned node v , among all assigned nodes, at level i of some eight-tree labeled $i + 1$ where v 's sibling is a free node.*

Proof. The lemma can be proved by induction on number of requests. It can be verified that the base case is true and if the two properties hold before a request arrives, the assignment and reassignment carried out by the assignment scheme maintain the two properties. \square

Lemma 4. *Assume that the total bandwidth of the assigned nodes is less than 2^h . For any i , $0 \leq i \leq h - 3$, when T^* contains an assigned node at level i , there is always a level- i free node in some eighth-tree labeled k , for $0 \leq k \leq i + 1$.*

Proof. This proof is similar to the proof of Lemma 1. We prove that if there is no level- i free node in any eighth-tree labeled k for $0 \leq k \leq i + 1$, then the total bandwidth of the assigned nodes is at least 2^h , which contradicts to the assumption.

Consider the eighth-trees labeled $i + 1$, by the assignment scheme, all assigned nodes in those eighth-trees must be at least at level i . As there is no level- i free node, the total bandwidth of the assigned nodes is at least $3 \cdot 2^{h-3}$. Consider the eighth-trees labeled i , all assigned nodes must be at least at level $(i - 1)$. By Lemma 3, all, except one, assigned nodes at level $(i - 1)$ find their siblings also assigned nodes. If that exception exists, there must be one assigned node at level $(i - 1)$ of T^* . Thus, the total bandwidth of the assigned nodes in the eighth-trees labeled i and the assigned node at level $(i - 1)$ of T^* (if any) is at least $3 \cdot 2^{h-3}$. Similarly, we consider the eighth-trees labeled k for $1 \leq k \leq i - 2$. With a similar argument making use of Lemma 3, we find that the total bandwidth of the assigned nodes in the eighth-trees labeled k and the assigned node at level $(k - 1)$ of T^* (if any)

is at least $3 \cdot 2^{h+k-i-2}$. Consider the six eighth-trees labeled 0. The total bandwidth of the assigned nodes is at least $6 \cdot 2^{h-i-2} = 3 \cdot 2^{h-i-1}$. Altogether, the total bandwidth of all assigned nodes is at least $3 \cdot 2^{h-3} + 3 \cdot 2^{h-3} + \sum_{k=1}^{i-2} 3 \cdot 2^{h+k-i-2} + 3 \cdot 2^{h-i-1} = 9 \cdot 2^{h-3} > 2^h$. \square

Theorem 3. *For the online tree node assignment problem, we have a 2-competitive algorithm using $3h/8 + 2$ trees and the cost of serving each request is at most one.*

3 Constant-competitive algorithms with constant number of trees

First, we give a $8/3$ -competitive algorithm using $11/4$ trees. Then, we modify the algorithm such that it uses $11/4 + (4/(3\alpha))$ trees, for any α where $0 < \alpha \leq 4/3$, and achieves a competitive ratio of $(4/3 + \alpha)$. Both algorithms are based on an extended concept of *compact configuration* of assigned nodes in trees [7], which is described below.

Assume that the available trees to the online algorithm are arranged on a line. For any two nodes u and v , where u is not an ancestor of v and vice versa, we say that u is on the *left* of v if u is in a tree which is on the left of the tree containing v , or there is a leaf in the subtree rooted at u which is on the left of a leaf in the subtree rooted at v ; otherwise, u is on the *right* of v . Level i of the trees is defined to be *compact* if all nodes of level i to the left of a blocked node of level i , which is blocked by an assigned node at level no more than i , are also blocked. We say that a configuration is compact if all levels of the trees are compact.

It is very costly to maintain a compact configuration after serving each request. By using a relaxed “compact” configuration with a constant number of trees, the amortized competitive ratio can be reduced to a constant. The idea of the relaxation is as follows: for each level i , there may exist more free nodes, which are immediate to the right of assigned nodes of level i , than the compact configuration. Such kind of free node can accommodate the following assignment request immediately without reassigning nodes at higher levels. Thus, there may be no extra cost (reassignment) after serving some request and the amortization of each request can be reduced to a constant by using some extra resources (the free node to the right of some assigned nodes may be wasted). Using this idea, we can achieve a $8/3$ -competitive algorithm with $11/4$ trees. Moreover, the competitive ratio can be further reduced by using a more completed relaxing technique, i.e., the assigned nodes at the same level may be not consecutive all the time, and we will compact them when needed. Thus, the amortized cost can be further reduced.

3.1 $8/3$ -competitive algorithm with $11/4$ trees

To design an assignment scheme with better performance, we make use of a “less compact” configuration, which is called the *almost-compact configuration*. This configuration stores odd-level and even-level assigned nodes separately. However, the way to store the two sets of assigned nodes is the same. For simplicity, we would show how even-level assigned nodes are stored only. The main idea in the almost-compact configuration is to allow more level- i free nodes immediately on the right of the rightmost level- i assigned node. Precisely, we allow at most seven of such free nodes. Note that a contiguous four level- i free nodes make up a level- $(i + 2)$ free node. Fig. 2 gives an example of an almost-compact configuration. The definition of a *almost-compact configuration* is that (1) for any two assigned nodes u and v at level ℓ_u and ℓ_v , respectively, we have u on the left of v if $\ell_u \leq \ell_v$, and (2) all, except one, free node at level j must be on the right of all assigned nodes at level i for $i \leq j$. The exception level- j free node must be on the left of all level- j assigned nodes. Note that condition (2) implicitly allows at most seven level- i free nodes immediate on the right of the the rightmost level- i assigned node that would be used to assign to level- i requests.

To describe the algorithm, we define a notation called the *level- i region*, which consists of all level- i assigned nodes and the following level- i free nodes. There is a level- i region for each level i . If there is no assigned nodes at level i , the level- i region may consist of only free level- i nodes or an empty region.

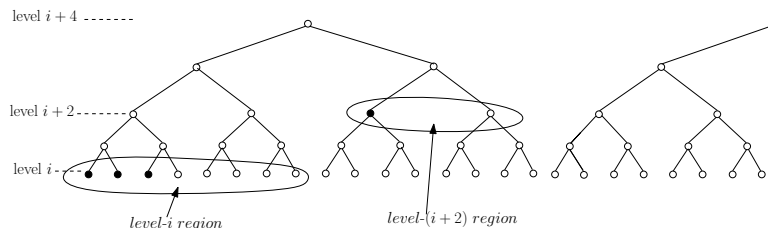


Figure 2: An example of an almost-compact configuration.

The details of the algorithm is given below.

Assignment request R of level i :

- Case A1.** If there is a level- i free node in the level- i region, assign R the leftmost such free node.
- Case A2.** If there is no level- i free node in the level- i region, find the non-empty level- j region G_j for the smallest $j > i$
 - (1) If G_j has no assigned node, i.e., the leftmost node in G_j is a free node, the free node is “divided” into four level- i free nodes, three level- k nodes for even k between $i + 2$ to $j - 2$. These free nodes are inserted to the corresponding level- i and level- k regions. The leftmost free node in the level- i region is assigned to R .
 - (2) Otherwise, release the leftmost assigned node u in G_j (which is reassigned later). The released node is divided and assigned as in step (1). An assignment request of level j is issued to find a free node for u .

Release request R of level i :

Release the assigned node for R , say v , and mark it free. If there is any level- i assigned node on the right of v , reassign the rightmost such assigned node to v .

- Case R1.** If the number of free nodes in the level- i region is less than eight, do nothing.
- Case R2.** If the number of free nodes in the level- i region is eight, find the level- j region G_j with an assigned node or with less than seven free nodes for the smallest $j > i$.
 - (3) If there are only free nodes in G_j , for even k from i to $j - 2$, the rightmost four level- k free nodes in the level- k region forms a level- $(k + 2)$ free node and that node is moved to the level- $(k + 2)$ region.
 - (4) Otherwise, perform step (3). In addition, “copy” the rightmost assigned node u in G_j to the newly formed free node on the leftmost position. and issue a release request for the “old” u , which is at the rightmost assigned node in G_j .

It can be seen that to maintain the almost-compact configuration, we may need cascading reassignment. We use an amortized analysis to bound the average number of assignments/reassignments for serving each request. The idea is that we pay a fixed credit for each request. If the credit is higher than the actual cost, the remaining credit is saved. If the credit is lower, the saved credit will be sufficient to pay the difference. We defined the *potential* of the almost-compact configuration, and

show that the saved credit is at least the potential and the potential is at least the different between the total credits paid and the actual cost.

The potential of an almost-compact configuration is $\sum_{\text{all even level } i} P_i$ where P_i is the potential of the level- i region, defined as follows.

Number of free nodes in the region	0	1	2	3	4	5	6	7
Potential	1	2/3	1/3	0	0	1/3	2/3	1

Before any request is sent, to start with, we need an almost-compact configuration with potential 0. We assign four level-0 free nodes to the level-0 region and three level- i free nodes to the level- i region for even $i > 0$.

The following lemma shows that if we pay a fixed credit of $4/3$ for serving each request, the saved credit is able to pay the actual cost of the algorithm for serving each request.

Lemma 5. *Let S_b and S_a be the potential of the almost-compact configuration before and after serves a request. We have $4/3 - (S_a - S_b)$, which is at least the actual cost spent by the algorithm. (The proof of this lemma is in Appendix.)*

Lemma 6. *Our algorithm in this section uses at most $11/4$ trees.*

Proof. We can implement the algorithm so that it uses the available trees in such a way that the even-level regions are occupying the trees from left to right and the odd-level regions from right to left. Then, it does not need to maintain free nodes at levels $h-2$, $h-1$ and h explicitly. The total bandwidth of the free nodes maintained is at most $7 \cdot \sum_{i=0}^{h-3} 2^i \leq 7 \cdot 2^{h-2}$. Together with the assigned nodes which have the total bandwidth at most 2^h , the algorithm needs to maintain nodes of total bandwidth at most $11 \cdot 2^{h-2} = (11/4) \cdot 2^h$, i.e., $11/4$ trees. \square

Theorem 4. *For the online tree node assignment, our algorithm in this section uses $11/4$ trees and it is $8/3$ -competitive.*

Proof. For a sequence with m_1 assignment requests and m_2 release requests, the cost of the optimal offline algorithm is at least m_1 , while the cost of our algorithm is at most $\frac{4}{3}(m_1+m_2) \leq \frac{4}{3}(2 \cdot m_1) = \frac{8}{3}m_1$. Thus, our algorithm using $11/4$ trees is $8/3$ -competitive. \square

3.2 $(4/3 + \alpha)$ -competitive algorithm with $11/4 + 4/(3\alpha)$ trees

To improve the competitive ratio of the algorithm in last section, we observe that we could be more lazy in tidying up the configuration when serving release requests. In this section, we define a less “tidy” configuration called the *semi-compact configuration*. Similar to the almost-compact configuration, the semi-compact configuration stores odd-level and even-level assigned nodes separately, but with the same method. Again, we will show how even-level assigned nodes are stored only. The semi-compact configuration also maintains level- i regions (as in the almost-compact configuration) for each even level i . The different between the two configurations is that the semi-compact configuration divides the level- i region into two contiguous parts, the *main region* on the left and the *gap region* on the right.

- The main region consists of assigned nodes and maybe some free nodes, but the number of free nodes is at most β times the number of assigned nodes, where $\beta \geq 1$ is a fixed parameter.
- The gap region consists of only free nodes and the number of free nodes is at most 7.

The details of the algorithm is given as follows.

Assignment request R of level i :

- Case A1.** If there is a level- i free node in the level- i main region, assign R the free node.
- Case A2.** If there is a level- i free node in the level- i gap region, assign R the leftmost free node, say u , and u is moved from the gap to the main region.
- Case A3** If there is no level- i free node in the level- i region, find the non-empty level- j region G_j for the smallest $j > i$
- (1) If G_j has no assigned node, i.e., the leftmost node is a free node, the free node is “divided” into four level- i free nodes, three level- k nodes for even-level k between $i + 2$ to $j - 2$. These free nodes are inserted to the corresponding level- i and level- k gap regions. The leftmost level- i free node is assigned to R . Similar to the case A2, the newly assigned node is moved from the gap to the main region.
 - (2) Otherwise, release the leftmost assigned node, say u , of G_j , which is reassigned later. The released node is divided and assigned as in step (1). An assignment request of level j is issued to find a free node for u .

Release request R of level i :

Release the assigned node for R .

- Case R1.** If the number of free nodes in the level- i main region is at most β times the number of assigned nodes, do nothing.
- Case R2.** If the number of free nodes in the level- i main region is more than β times the number of assigned nodes, compact the main region into contiguous assigned nodes on the left by reassignments. The free nodes are moved to the gap regions.
- (3) If the number of free nodes in the gap region is at most 7, do nothing.
 - (4) If the number of free nodes in the gap region is more than 7, let the number be in the form $4x + y$ where x and y are integers and $x > 1$ and $4 \leq y \leq 7$. Group the rightmost $4x$ free nodes into x level- $(i + 2)$ free nodes. The x level- $(i + 2)$ free nodes are moved to the level- $(i + 2)$ main region in a way that are considered as x release requests.

We use an amortized analysis, similar to that in last section, to bound the average number of assignments/reassignments needed for serving each assignment or release request. The credits paid for an assignment request is $4/3$ and a release request is α where $\alpha = 4/(3\beta) \leq 4/3$ as $\beta \geq 1$. The potential of a level- i main region is α times the number of free nodes in the main region. The potential of the level- i gap region is defined as follows.

Number of free nodes in the gap region	0	1	2	3	4	5	6	7
Potential	1	2/3	1/3	0	0	$\alpha/4$	$\alpha/2$	$3\alpha/4$

The potential of a semi-compact configuration is the sum of all potential of the level- i main and gap regions for all level i . The initial semi-compact configuration has four level-0 free nodes in the level-0 gap region, and three level- i free nodes in the level- i gap region for all other $i > 0$. All main regions are empty. The initial potential is 0.

The following lemma shows that the saved credit is able to pay the actual cost of the algorithm for serving each request.

Lemma 7. *Let S_b and S_a be the potential of the semi-compact configuration before and after serving a request. The actual cost to serve an assignment request is at most $4/3 - (S_a - S_b)$ and a release request is at most $\alpha - (S_a - S_b)$. (The proof is in Appendix.)*

Lemma 8. *Our algorithm in this section uses at most $11/4 + 4/(3\alpha)$ trees. (The proof is in Appendix.)*

Theorem 5. *For the online tree node assignment problem, our algorithm in this section is $(4/3 + \alpha)$ -competitive and it uses at most $11/4 + 4/(3\alpha)$ trees, for any α where $0 < \alpha \leq 4/3$.*

References

- [1] G. S. Brodal, E. D. Demaine, and J. I. Munro. Fast allocation and deallocation with an improved buddy system. *Acta Inf.*, 41(4-5):273–291, 2005.
- [2] F. Y. L. Chin, H.-F. Ting, and Y. Zhang. A constant-competitive algorithm for online ovfs code assignment. To appear in *Algorithmica*. In *Proceedings of the 18th International Symposium on Algorithms and Computation (ISAAC)*, pages 452–463, 2007.
- [3] F. Y. L. Chin, H. F. Ting, Y. Zhang. Constant-Competitive Tree Node Assignment manuscript.
- [4] F. Y. L. Chin, Y. Zhang, and H. Zhu. Online OVFS code assignment with resource augmentation. In *Proceedings of the Third International Conference on Algorithmic Aspects in Information and Management (AAIM)*, pages 191–200, 2007.
- [5] D. C. Defoe, S. R. Cholleti, and R. Cytron. Upper bound for defragmenting buddy heaps. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 222–229, 2005.
- [6] S. Dutt and J. P. Hayes. Subcube allocation in hypercube computers. *IEEE Trans. Computers*, 40(3):341–352, 1991.
- [7] T. Erlebach, R. Jacob, M. Mihalák, M. Nunkesser, G. Szabó, and P. Widmayer. An algorithmic view on OVFS code assignment. *Algorithmica*, 47(3):269–298, 2007.
- [8] M. Forisek, B. Katreniak, J. Katreniaková, R. Kralovic, R. Královic, V. Koutný, D. Pardubská, T. Plachetka, and B. Rován. Online bandwidth allocation. In *Proceedings of the 15th Annual European Symposium on Algorithms (ESA)*, pages 546–557, 2007.
- [9] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *J. ACM*, 47(4):617–643, 2000.
- [10] K. C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, 1965.
- [11] D. E. Knuth. *The Art of COmputer Programming, Volumn 1: Fundamental Algorithms*. Addison-Wesley, 1975.
- [12] X.-Y. Li and P.-J. Wan. Theoretically good distributed CDMA/OVFS code assignment for wireless ad hoc networks. In *Proceedings of the 11th Annual International Conference of Computing and Combinatorics (COCOON)*, pages 126–135, 2005.
- [13] T. Minn and K.-Y. Siu. Dynamic assignment of orthogonal variable-spreading-factor codes in W-CDMA. *IEEE Journal on Selected Areas in Communications*, 18(8):1429–1440, 2000.
- [14] S. Miyazaki and K. Okamoto. Improving the competitive ratio of the online OVFS code assignment problem. In *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC)*, pages 64–76, 2008.
- [15] A. N. Rouskas and D. N. Skoutas. OVFS codes assignment and reassignment at the forward link of W-CDMA 3G systems. In *Proceedings of the 13th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, volume 5, pages 2404–2408, 2002.
- [16] T. Erlebach, R. Jacob and Marco Tomamichel. Algorithmische Aspekte von OVFS Code Assignment mit Schwerpunkt auf Offline Code Assignment. Student thesis at ETH Zürich.
- [17] P.-J. Wan, X.-Y. Li, and O. Frieder. OVFS-CDMA code assignment in wireless ad hoc networks. *Algorithmica*, 49(4):264–285, 2007.

Appendix

Proof of Lemma 2

Proof. We prove by induction that at the end of Step i , for $0 \leq i \leq h - 2$,

1. the number of level- $(i + 1)$ nodes blocked is $(i + 2) \cdot 2^{h-i-2}$,
2. in the subtree rooted at each level- $(i + 1)$ blocked node, there is exactly one assigned node.

For the base case, consider Step 0. The adversary can choose to release requests such that no two released nodes are within the same subtree rooted at a level 1 node. Thus, the remaining level-0 assigned nodes block 2^{h-1} level-1 nodes. Both statements of the induction are true.

Assume the induction is true at the end of Step i , we consider Step $i + 1$. We describe a release strategy of the adversary such that the two statements in the induction are true. The strategy is designed according to the current configuration of how assigned nodes are grouped in the subtrees rooted at the level- $(i + 2)$ blocked nodes. By the second statement of the induction for Step i , there may be one or two, but no more, assigned nodes in the subtree rooted at a level- $(i + 2)$ blocked node. In the case of two, one of the assigned nodes has to be released.

The details of the release strategy are defined as follows. We first construct an undirected graph $G = (V, E)$ based on the configuration of how assigned nodes are grouped in the subtrees rooted at the level- $(i + 2)$ blocked nodes. The vertex set V consists of $i + 1$ vertices, v_0, v_1, \dots, v_i , where v_k represents the set of assigned nodes at level k . For each subtree rooted at a level- $(i + 2)$ blocked node that contains two assigned nodes, say at levels s and t , E contains an edge (v_s, v_t) . We arbitrarily pair up those subtrees rooted at a level- $(i + 2)$ dead nodes that contain only one assigned node. For each pair of the subtrees, let the corresponding assigned nodes be at levels s' and t' , E contains an edge $(v_{s'}, v_{t'})$. Note that the resulting graph G may consist of multiple edges between a pair of vertices and self-loops of a vertex. Yet, G is an Eulerian graph because the number of assigned nodes at each level is even and hence the degree of each vertex in G is even. The release strategy takes any Eulerian cycle of G and decides which assigned nodes should be released. We trace the cycle in one direction and assume the edges in the cycle are directed in that direction. Thus, for each directed edge (v_s, v_t) in the cycle, let T_1 and T_2 be the corresponding subtrees rooted at some level- $(i + 2)$ blocked nodes with assigned nodes at levels s and t , respectively. T_1 and T_2 can be the same subtree. The release strategy releases the level- s assigned node in T_1 and keeps the level- t assigned node in T_2 .

For each vertex v_j in G , the degree of v_j is the number of assigned nodes at level j . In a directed Eulerian cycle of G , the number of times that it enters v_j and the number of times that it leaves of v_j are same. Thus, after executing the release strategy, the number of level- j assigned nodes released and the number of level- j assigned nodes remain are same, which is half as that remained in the previous step, i.e., 2^{h-i-2} if $j = 0$ and 2^{h-i-3} if $1 \leq j \leq i + 1$. Moreover, for each subtree rooted at a level- $(i + 2)$ blocked node with two assigned nodes, one of the assigned nodes is released. As a result, the two statements of the induction are true, and hence the induction completes. \square

Proof of Lemma 5

Proof. We consider the four cases, A1, A2, R1, and R2 where the algorithm handles a request.

For case A1, the actual cost is one and the change in potential of the level- i region, as well as the configuration, is at most $1/3$. Thus $4/3 - (S_a - S_b) \geq 4/3 - 1/3 = 1$ which is at least the actual cost.

For case A2(1), the actual cost is one. For the level- i region, the change in potential is -1 . For the level- k region for even k from $i + 2$ to $j - 2$, the new potential is 0, so the change is at most 0. For the level- j region, the change in potential is at most $1/3$. Again, we have $4/3 - (S_a - S_b) \geq 4/3 - (-1 + 1/3) = 2$ which is greater than the actual cost.

For case A2(2), the actual cost is one plus the actual cost of an extra assignment request of level- j . As in case A2(1), the change in potential for the level- k region for even k from i to $j - 2$ is -1 , which is sufficient to pay the first assignment. Then the fixed cost $4/3$ together with the change in potential of the level- k for even $k \geq j$ should be able to pay the extra assignment request of level- j .

For case R1, the actual cost is at most one. The change in potential of the level- i region is at most $+1/3$. Thus $4/3 - (S_a - S_b) \geq 4/3 - 1/3 = 1$, which is at least the actual cost.

For case R2(3), the actual cost is at most one. The change in potential of the level- i region is -1 . The new potential of level- k region for even k from $i + 2$ to $j - 2$ is 0, so the change is at most 0. The change in potential of the level- j region is at most $1/3$. Thus $4/3 - (S_a - S_b) \geq 4/3 - (-1 + 1/3) = 2$, which is greater than the actual cost.

For case R2(4), the actual cost is one plus the actual cost of an extra release request of level- j . As in case R2(3), the change in potential for the level- k region for even k from i to $j - 2$ is at most -1 , which is sufficient to pay the reassignment at level i . Then the fixed cost $4/3$ together with the change in potential of the level- k for even $k \geq j$ should be able to pay the release request at level- j . \square

Proof of Lemma 7

Proof. We consider the five cases, A1, A2, A3, R1, and R2 where the algorithm handles a request.

For case A1, the actual cost is 1 and the change in potential (from the level- i main region) is $-\alpha$. We have $4/3 - (S_a - S_b) = 4/3 + \alpha > 1$.

For case A2, the actual cost is 1 and the change in potential (from the level- i gap region) is at most $1/3$ as $\alpha/4 \leq 1/3$. We have $4/3 - (S_a - S_b) \geq 4/3 - 1/3 = 1$.

For case A3(1), the actual cost is 1. For the level- i region, the change in potential is -1 . For the level- k region for even k from $i + 2$ to $j - 2$, the new potential is 0, so the change is at most 0. For the level- j region G_j , the change in potential is at most $\max\{\alpha, 1/3\} \leq 4/3$, depending on whether the leftmost free node is from the main or the gap region. We have $4/3 - (S_a - S_b) \geq 4/3 - (-1 + 4/3) = 1$.

For case A3(2), the actual cost is 1 plus the actual cost of an extra assignment request of level- j . As in case A3(1), the change in potential for the level- k region for even k from i to $j - 2$ is -1 , which is sufficient to pay the first assignment of cost 1. Then the fixed cost $4/3$ together with the change in potential of the level- k for even $k \geq j$ should be able to pay the extra assignment request of level- j .

For case R1, the actual cost is 0. The change in potential (from the level- i main region) is at most α . We have $\alpha - (S_a - S_b) \geq \alpha - \alpha = 0$.

For case R2(3), let m_a and m_f be the number of assigned and free nodes in the level- i main region after the assigned node is released. The actual cost is the number of reassignment in the main region, which is at most the number of assigned nodes, i.e., m_a . For the level- i main region, the potential before is $(m_f - 1)\alpha$ and the potential after the compaction is 0 (as with no free node). The change is $-(m_f - 1)\alpha$. The change in potential of the level- i gap region is at most $m_f\alpha/4$. Therefore, $\alpha - (S_a - S_b) \geq \alpha - (-(m_f - 1)\alpha + m_f\alpha/4) = 3m_f\alpha/4 = m_f/\beta > m_a$ since compaction is triggered in case R2.

For case R2(4), let m_a and m_f be the number of assigned and free nodes in the level- i main region after the node is released. The actual cost is the number of reassignments in the main region, which is at most m_a , plus the actual cost of x extra release requests of level- $(i + 2)$. As in case R2(3), the change in potential in the main region is $-(m_f - 1)\alpha$. The change in potential in the gap region is at most $(m_f - 4x)\alpha/4$ as the $4x$ free nodes will be grouped and moved to level- $(i + 2)$. Therefore, the change of potential in the level- i main and gap regions is at least $-(m_f - 1)\alpha + (m_f - 4x)\alpha/4 = -3m_f\alpha/4 - (x - 1)\alpha < -m_a - (x - 1)\alpha$. Note that the value $-m_a$ covers the actual cost in the compaction of the level- i main region. Together with the credit α for the release request, we have $x\alpha$ remaining potential to support the extra x release requests at level- j , which is sufficient. \square

Proof of Lemma 8

Proof. We can implement the algorithm so that it uses the available trees in such a way that the even-level regions are occupying the trees from left to right and the odd-level regions from right to left. Then, it does not need to maintain the gap regions at levels $h - 2$ and $h - 1$ explicitly. The total bandwidth of the free nodes in the gap regions is at most $7 \cdot \sum_{i=0}^{h-3} 2^i \leq 7 \cdot 2^{h-2}$. The total bandwidth of the assigned nodes (in the main regions) is at most 2^h . The total bandwidth of the free nodes in the main regions is at most $\beta \cdot 2^h = 4 \cdot 2^h / (3\alpha)$. Hence, the algorithm needs to maintain nodes of total bandwidth at most $(11/4 + 4/(3\alpha))2^h$, i.e., the algorithm is using at most $11/4 + 4/(3\alpha)$ trees. \square