Theoretical Computer Science 518 (2014) 10–21

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Constant-competitive tree node assignment

Yong Zhang [a,b,*,1], Francis Y.L. Chin [b,2], Hing-Fung Ting [b,3]

[a] College of Mathematics and Computer Science, Hebei University, China
[b] Department of Computer Science, The University of Hong Kong, Hong Kong

ARTICLE INFO

Article history:
Received 15 September 2010
Received in revised form 1 March 2013
Accepted 8 May 2013
Communicated by T. Erlebach

Keywords:
Online algorithms
Tree node assignment
Competitive ratio

ABSTRACT

In this paper, we study the online tree node assignment problem, which is a generalization of the well studied OVSF code assignment problem. Assigned nodes in a complete binary tree must follow the rule that each leaf-to-root path must contain at most one assigned node. At times, it is necessary to swap assigned nodes with unassigned nodes in order to accommodate some new node assignment. The target of this problem is to minimize the number of swaps in satisfying a sequence of node assignments and releases.

This problem is fundamental, not only to the OVSF code assignment, but also to other applications, such as buddy memory allocation and hypercube subcube allocation. All the previous solutions to this problem are based on a sorted and compact configuration by assigning the nodes linearly and level by level, ignoring the intrinsic tree property in their assignments.

Our contributions are: (1) give the concept of safe assignment, which is proved to be unique for any fixed set of node-assignment requests; (2) an 8-competitive algorithm by holding the safe assignment; and (3) an improved 6-competitive variant of this algorithm. Our algorithms are simple and easy to implement and our contributions represent meaningful improvements over recent results.

1. Introduction

The problem we study in this paper involves the assignment and release of nodes in a complete binary tree when faced with a sequence of requests for either an assignment of a node at specified level of the tree, or a release of an assigned node of the tree. The only rule which dictates the assignments/releases is that the tree remains in a *legal configuration* where no two assigned nodes lie on a single path from the root to a leaf. Fig. 1 is an example of a legal tree with assigned nodes darkened and marked as c, d, e, g and i.

The requirement of our problem is to accommodate each request if at all possible and, in order to accommodate a new request, it might be necessary to *swap* nodes to "make room" for the new request, where *swap* means changing the position of an assigned node with an unassigned node at the same level. For example, in Fig. 1, to accommodate a request for node assignment at level 2, we could first change the assignment by swapping node c with node f and then satisfy the request with the assignment of node a at level 2. Alternatively, we could swap node g and node b and then assign node j at level 2 to satisfy the request. Since each swap represents processing overhead, we have the optimization objective of designing algorithms to solve this tree assignment problem so as to minimize the number of swaps.

* Corresponding author at: Department of Computer Science, The University of Hong Kong, Hong Kong.
E-mail addresses: yzhang@cs.hku.hk (Y. Zhang), chin@cs.hku.hk (F.Y.L. Chin), hfting@cs.hku.hk (H.-F. Ting).

[1] Research supported by NSFC (No. 11171086) and HKU Small Project Funding 7176218.
[2] Research supported by HK RGC grant HKU-711709E.
[3] Research supported by HKU Small Project Funding 7176115.

0304-3975/$ – see front matter
http://dx.doi.org/10.1016/j.tcs.2013.05.014

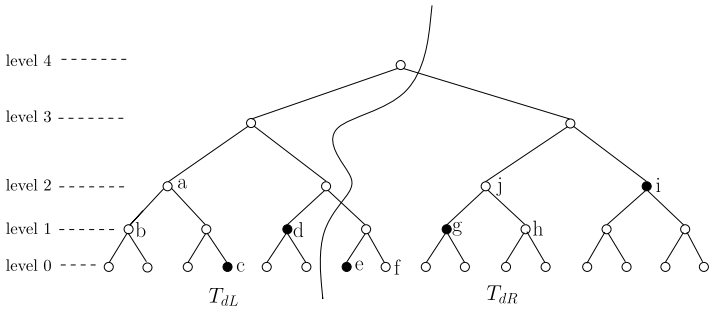

Fig. 1. Example of legal configuration, solid circles are assigned nodes.

Table 1
Application-specific problems and how the node and the swap are interpreted.

Problem	Node at level l	Swap
OVSF code assignment	Code of frequency bandwidth 2^l	Code reassignment
Buddy memory allocation	Memory block of size 2^l	Memory reallocation
Hypercube subcube allocation	Subcube of 2^l processors	Subcube migration

As it turns out, a solution to our problem, given its fundamental formulation, can be used to solve a variety of application-specific problems including the Orthogonal Variable Spreading Factor (OVSF) code assignment problem [2,4,8,11,14], the buddy memory allocation problem [1,6,12,13] and the hypercube subcube allocation problem [7]. The main difference between these problems is how the node at level l and the swap operation are interpreted (see Table 1).

As with other scheduling-type problems, there naturally arises an off-line and an online version of this problem. In the off-line version, the sequence of requests is made known to the algorithm at the outset, whereas in the online version, the algorithm must process each request without the benefit of any information about future requests. The off-line version of the tree node assignment problem is NP-hard [9], so the approach is to produce a good heuristic algorithm, whose performance is measured by the ratio of the cost of the heuristic algorithm to the cost of an optimal off-line algorithm. The ratio in case of the online problem (the off-line problem) is called the *competitive ratio* (*approximation ratio*).

There is a variety of ways to define the cost of an algorithm. For example, cost could simply be the total number of assignments and swaps done by the algorithm, with zero cost for each node release. This cost definition is justified since, for most applications, it is expensive to set up a node assignment while there is little cost associated with a node release (e.g. terminating communications, freeing some memory, or releasing the hypercube subcube for another task) and a swap can be considered equivalent to a node assignment plus a node release. In the actual application, higher level node assignment/swap for buddy memory allocation and hypercube subcube allocation might be more costly because larger size of the memory blocks and subcube are involved. However, a uniform cost for each level node assignment/swap is justified if the data transfer cost is considered to be negligible when compared with the high setup cost.

In this paper, we focus on the online problem and we define cost to be the total number of assignments and swaps done by the algorithm. For the online problem, Erlebach et al. [8] 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. With resource augmentation in the form of using a bigger tree, it is possible to have constant-competitive algorithms. Erlebach et al. [8] first gave a 4-competitive algorithm with two trees, Chin et al. [5] gave a 5-competitive algorithm with 9/8 trees. By balancing the performance ratio and resource augmentation, Chan et al. [3] gave a 2-competitive algorithm with $3h/8+2$ trees; a 8/3-competitive algorithm with 11/4 trees; and a general $(4/3+\alpha)$-competitive algorithm with $(11/4+4/(3\alpha))$ trees, for any $0<\alpha\leqslant 4/3$. Without resource augmentation, Forisek et al. [10] first gave a constant-competitive algorithm, but without deriving the exact value of the constant. According to their scheme, each node release may still cost $O(h)$ swaps, but with constant value potential defined on some nodes, a constant-competitive algorithm could be achieved. A 10-competitive algorithm [4] was also derived based on a lazy approach to group assigned nodes at some consecutive levels together. Miyazaki and Okamoto [14] introduced a 7-competitive algorithm and proved a lower bound of 2 on the competitive ratio.

All of the previous work has been based on a sorted and compact legal configuration, where all assigned nodes are sorted according to their levels in non-decreasing order from left to right in the tree and are pushed to the left as much as possible. In these methods, nodes are "packed" level by level, ignoring the intrinsic tree property in their assignments. By studying properties for a "good" legal configuration, one of our main contributions is to define a novel configuration, called *dense* configurations (to be defined in Section 2) which is not only a generalization of the sorted and compact configuration, but can fully utilize the capacity of the tree and allow new requests to be served easily. In fact, the sorted and compact configuration is one of two extreme cases of the dense configurations that require swaps in order to maintain the "dense" property. In this paper, we define a *safe* configuration, which is dense and easy to maintain, and does not have the problem of high cost in the sorted and compact configuration.