Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 67 (2007) 674–694
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Self-tuning reactive diffracting trees☆

Phuong Hoai Ha, Marina Papatriantafilou, Philippas Tsigas*

Department of Computer Science and Engineering, Chalmers University of Technology, S-412 96 Göteborg, Sweden

Received 23 February 2006; received in revised form 14 November 2006; accepted 23 January 2007
Available online 18 March 2007

Abstract

Reactive diffracting trees are efficient distributed objects that support synchronization, by distributing sets of memory accesses to different memory banks in a coordinated manner. They adjust their size in order to retain their efficiency in the presence of different contention levels. Their adjustment is sensitive to parameters that have to be manually determined after experimentation. Since these parameters depend on the application as well as on the system configuration and load, determining their optimal values is difficult in practice. Moreover, the adjustments are done one level at a time, hence the cost of multi-level adjustments can be high.

This paper presents a new method for reactive diffracting trees, without the need of hand-tuned parameters. The new *self-tuning* trees (ST-trees) can balance, in an online manner, the trade-off between the tree-traversal latency and the latency due to contention on accessing the leaf nodes (i.e. the nodes where the desirable computation takes place). Moreover, the paper presents a data structure that enables the trees to grow or shrink by several levels in one adjustment step. The behavior of the reactive diffracting trees is illustrated in the paper via experiments performed on a well-known ccNUMA multiprocessor system. The experiments study the new self-tuning trees, also in connection with the original hand-tuned reactive diffracting trees. The experiments have showed that the new self-tuning trees are efficient, and that they react in the same way (i.e. select the same tree depth for the same contention level) as the hand-tuned trees, while they are able to adjust quicker than the latter (as they are able to grow or shrink by several levels in one adjustment step).

Keywords: Synchronization; Concurrent programming; Dynamic data structures; Distributed data structures; Trees

1. Introduction

It is hard to design distributed synchronization objects that perform efficiently over a wide range of contention conditions. Typically, simple centralized data structures are sufficient for low concurrency levels, for instance test-and-test-and-set locks with exponential backoff [1,3,12,17]; however, at higher levels of concurrency, sophisticated data structures are needed to reduce contention by distributing concurrent memory accesses to different banks, for instance queue-locks [3,12,21], combining trees [29,11], counting pyramids [27,28], combining funnels [25], counting networks [4,5] and diffracting trees [24,23].

Whereas queue-locks aim at reducing contention on locks generally, the other sophisticated data structures focus on specific problems such as counting and balancing. Combining trees [29,11] implement low-contention `fetch-and-`Φ operations by combining requests along paths upward to their root and subsequently distributing results downwards to their leaves. The idea has been developed to counting pyramids [27,28] that allow nodes to randomly forward their request to a node on the next higher level and also allow processors to select their initial level according to their request frequency. A similar idea has been used to develop combining funnels [25].

Opposite to the idea of combining requests, *diffracting trees* [24,23] reduce contention for counting problems by distributing requests (tokens) downward to the leaves; each leaf works as a counter, assigning numbers to the tokens that go through it, in a way such that the set of numbers received by all the tokens is the same as the set of numbers that would have been issued by a single counter. Another approach for counting problems is counting networks [4,5], which may have lower contention at each node, since they can have multiple entry points for the tokens. Linearizability [16] in counting networks with and without timing assumptions has been studied in [20,19].

☆ Extended version of an article published in the Proceedings of the 8th International Conference on Principles of Distributed Systems (OPODIS '04), Lecture Notes in Computer Science, vol. 3544, Springer, Berlin, pp. 213–228.

* Corresponding author. Fax: +46 31 772 3663.
E-mail address: tsigas@cs.chalmers.se (P. Tsigas).

0743-7315/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2007.01.011

Empirical studies using Proteus [7], a multiprocessor simulator, have showed that diffracting trees are superior to counting networks and combining trees under high-contention [24].

Diffracting trees essentially distribute concurrent memory accesses to different banks, in a coordinated manner. Each process(or) accessing the tree can be considered as leading a *token* that follows a path with nodes from the root to a leaf. Each node receives tokens from its single input (coming from its parent node) and sends tokens to its outputs. The node is called a *balancer* and acts as a *toggle mechanism*, which, given a stream of input tokens, alternately forwards them to its outputs, from left to right (i.e. send them to the left and right child nodes, respectively). The result is an even distribution of tokens at the leaves. Diffracting trees have been introduced for *counting problems*, and hence the leaves are counters, which can be implemented by shared variables whose access is coordinated via, e.g. some queue-lock-based method. Yet the fixed-size diffracting tree is optimal only for a small range of contention levels. To solve this problem, Della-Libera and Shavit proposed *reactive diffracting trees*, where nodes can shrink (to a single counter/leaf) or grow (to subtrees with counters as leaves) according to their local contention [8]. The idea is to be able to have small traversal latencies (i.e. short paths of nodes from the root to the leaves) in low-load situations, and also larger throughput in high-load situations (by distributing the tokens among a larger set of leaf counters).

The reactive diffracting trees as proposed in [8], use a set of *hand-tuned* parameters to make the reactive decisions; these parameters are the folding/unfolding-thresholds and the time-interval for consecutive reactions. The parameter values depend on the multiprocessor system in use, the applications using the data structure and, in a multiprogramming environment, the system utilization by the other concurrent programs. The parameters must be manually tuned using experimentation and information that is not easily available (e.g. future contention characteristics). Besides, the tree can shrink or grow by only one level at a time, making multi-level adjustments costly.

The main challenge in designing reactive objects in multiprocessor/multiprogramming systems is to deal with unpredictable changes in the execution environment. Therefore, reactive schemes using *fixed* parameters cannot be an optimal approach in *dynamic* environments such as multiprogramming systems. An efficient reactive object should not rely on hand-tuned parameters and should react fast. As was also mentioned in [8], it is worth investigating other ways of making reaction–decisions, besides the hand-tuned parameter-based method presented in that article.

In this work we show that it is possible to construct such efficient reactive objects that are self-tuning. In particular, we present a tree-type distributed data object that has the same semantics as the reactive diffracting tree and can react without the need of manual tuning. The new self-tuning tree (*ST-tree*), like the former, hand-tuned tree, is aimed for applications which need such concurrent data objects, to distribute concurrent memory accesses to different memory- or computation-banks in a coordinated manner (i.e. the leaves can be computing some other function, instead of counting). Since the latter has been introduced in the context of counting problems, the same context is kept in this paper. To circumvent the need of hand-tuned parameters, we analyze the problem of balancing the trade-off between the two key sources of latency for each token—namely latency due to traversal of the path of intermediate nodes (balancers) and latency due to contention to access a counter node (leaf)—as an online decision problem and subsequently design an efficient online solution. Moreover, we design a new data-structure that enables low-overhead multi-level adjustment: it can shrink and grow by multiple-levels at a single adjustment step, allowing for higher efficiency in performing the adjustments instructed by the reactive decisions. We also present here an extended experimental study on a commercial ccNUMA multiprocessor. The study showed that the new self-tuning trees are efficient, that they decide to react in the same way (i.e. select the same tree depth for the same contention levels) as the original hand-tuned trees, while they are able to adjust quicker than the latter, as they are able to grow or shrink by several levels in one adjustment step.

The rest of the paper is organized as follows. The next section provides the description of the problem and some background information on synchronization and online methods. Section 3 presents the data-structure and the algorithm for the coordination in the ST-trees. In Section 4 we describe the online method to make the reactive decisions in ST-trees, including the analysis that justifies the decision criteria. In Section 5 we give detailed pseudocode and description for the implementation of the ST-trees. Section 6 provides the correctness proof of the ST-trees construction. Section 7 presents an experimental study of the ST-trees, in connection with the original hand-tuned reactive diffracting tree on the SGI Origin2000 platform, and elaborates on a number of properties of the ST-trees. Finally, Section 8 concludes this paper.

2. Problem description and background

The definitions for (reactive) diffracting trees are well-known from the articles that introduced them [24,23,8]. For reasons of self-containment, we include here the definitions of the problems and the terms used throughout the paper.

2.1. Diffracting and reactive-diffracting trees

A $(f_{\text{in}}, f_{\text{out}})$-*balancer* is a computing element receiving tokens through its f_{in} *input wires,* and forwarding them to its f_{out} *output wires*; f_{in} and f_{out} are the *fan-in* and *fan-out* of the balancer, respectively. Intuitively, a balancer is like a toggle mechanism which, given a stream of input tokens (which may arrive on the balancer's input wires at arbitrary times), alternately forwards them to its output wires, thus balancing the number of tokens that traverse the balancer. The *state* of a balancer at a given time is the collection of tokens on each one of its input and output wires. A *balancing network* [4,5], is a network of balancers, where pairs of balancers (B_i, B_j) are connected by having some output wire(s) of B_i connect with some input wire(s) of B_j. The input (rep. output) wires of balancers

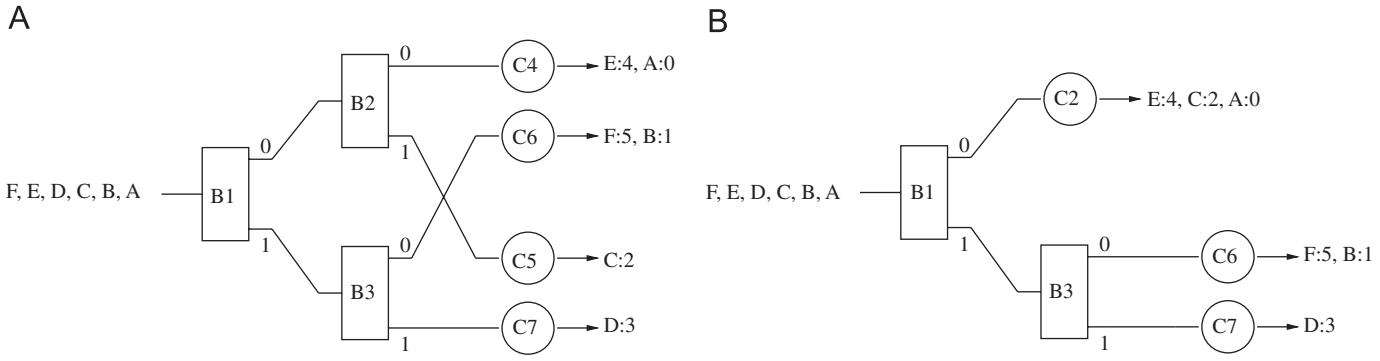

Fig. 1. A diffracting tree (A) and a reactive diffracting tree (B).

that are not connected with other balancers' wires form the input (resp. output) wires of the network. In a shared-memory system implementation, balancers typically are records, while wires are pointers from one record to another, having tokens (processes) visiting them asynchronously.

A balancing network satisfies the *step property*, if when there are no tokens present inside the network and if out_i denotes the number of tokens that have been output at the network's output wire i, $0 \leqslant out_i - out_j \leqslant 1$ for any pair i and j of output wires such that $i < j$ (i.e. if one draws the tokens that have exited from each output wire as a stack of boxes, the combined outcome will have the shape of a single step).

The *counting tree* [24,23] is a special type of a balancing network consisting of (1, 2)-*balancers*, interconnected as follows: the input wire of the root balancer, which is the single balancer on level 0 of the tree, is the entry point for the tokens that enter the network; each one of the two output wires of the root is connected to the input wire of the corresponding child-balancer at level 1 of the tree; the above is repeated recursively, until the leaves of the tree. The output wires are enumerated from left to right so that the output wires extending from the left or 0 (resp. right, or 1) wire of the root form the even (resp. odd) output wires of the tree. It has been shown [24,23] that the counting tree satisfies the step property. Such a tree can compute a function on the tokens that have traversed it (e.g. enumerate the tokens), by attaching corresponding computing elements (e.g. counters) to each of its output wires; each of these counters associate numbers to tokens that exit from it, in a way such that the sequence of tokens that exit through output wire i of the tree receive numbers $f(i)$, $f(i + w_{\text{out}})$, $f(i + 2w_{\text{out}})$, $f(i + 3w_{\text{out}})$, ..., where w_{out} is the total number of output wires of the *counting tree* and f is an arbitrary function, which may be a time-consuming function. For counting problems, $f(k)$ returns k.

Diffracting trees [24,23] are an implementation of such trees, where the tokens entering each balancer make use of an *elimination method* [22], that has the potential to evenly balance each pair of incoming tokens (and direct them left and right) without accessing the (single) toggle-bit (that can otherwise be used in order to implement such a balancer). Diffracting trees have been introduced in the context of *counting problems*, and hence the leaves are counters, which can be implemented by shared variables whose access is coordinated via, e.g. some queue-lock-based method. The same context is kept in this paper.

Fig. 1(A) depicts a diffracting counting tree. In a general execution, leaves $C4$, $C6$, $C5$ and $C7$ return values $f(4k)$, $f(4k + 1)$, $f(4k + 2)$ and $f(4k + 3)$, respectively, where k is the number of processors that have visited the corresponding leaf and f in the counting problem context returns k. Tokens passing one of these leaves receive integers i, $i+4$, $i+2*4$, ... where i is the initial value of the leaf. An example execution is as follows. Assume that the toggle-bits of $B1$, $B2$ and $B3$ are 0 at the beginning. Assume also that tokens A, B, C, D, E, F arrive in that order. When visiting $B1$, token A switches $B1$'s toggle-bit from 0 to 1 before going to $B2$ (branch 0 of $B1$). Then, it switches $B2$'s toggle-bit to 1 before going to $C4$ and getting value 0. Similarly, token B, when visiting $B1$, switches $B1$'s toggle-bit from 1 back to 0 before going to $B3$ (branch 1 of $B1$). Then, it switches $B3$'s toggle-bit to 1 before going to $C6$ and getting value 1. Consequently, tokens A, B, C, D, E and F receive integers 0, 1, 2, 3, 4 and 5, respectively.

Della-Libera and Shavit in [8] report, using a detailed experimental study, that (i) under low-load scenarios the throughput of diffracting trees is significantly higher when the tree is of lower depth (as the tokens do not need to traverse a long path of balancers from the root to the leaves before exiting); and that (ii) in high-load scenarios the throughput is significantly higher when the tree is deeper, as the tokens can exit concurrently via a larger number of leaves (i.e. they do not need to spend time competing with many other tokens for getting counted at the leaves). Having manifested this trade-off, they suggested to extend the trees into *reactive diffracting trees*, whose depth can be adjusted according to the load, aiming at balancing the trade-off, in order to maximize throughput by optimizing the tokens' latency which is due to (i) traversal of balancers and (ii) competition (contention) at the leaves/counters. Assuming that the maximum contention in the tree is bounded by the number of processors in the system (P), the maximum depth of the tree, in order to allow for minimum contention at the leafs, is bounded by $\log P$. Similarly, for the case of minimum contention, the tree should better degenerate to a single counter.

Note that the object is distributed, hence, *decisions for reaction* are to be made locally: each leaf (i.e. the process/token

visiting it) should be able to decide to shrink or grow, according to its local contention, in order to attain optimized performance/throughput [1] of the tree [8].

In response to a reactive decision the tree is *adjusted*. Such adjustments can cause situations when the tree is *irregular* (i.e. not all its leaves are at the same level). In order to count tokens correctly (i.e. to assign numbers to exiting tokens), the counters at the leaves of an irregular tree have to take into account their level at the tree. In particular, a counter/leaf C at level l with initial value $init(C)$ must issue values $init(C), init(C) + 2^l, \ldots, init(C) + i * 2^l, \ldots$ for the tokens going through it.

As an example consider trees (A) and (B) in Fig. 1, which illustrate a shrinkage adjustment of a reactive diffracting tree. Assume at the beginning the tree has the shape of (A). If the contention on leaves $C4$ and $C5$ is low (e.g. due to long latency for a token to go from the root $B1$ to these leaves), as the nodes along the path can be stored in remote memory banks in a non-uniform-memory-access (NUMA) system, the subtree rooted at $B2$ decides to react and the adjustment causes that subtree to shrink into leaf $C2$, as depicted in tree (B). After that, if tokens A, B, C, D, E and F sequentially traverse the tree (B), three tokens, i.e. A, C and E will visit leaf $C2$, which issues numbers $0, 2, 4, \ldots$.

To summarize, a counting/diffracting tree construction should provide processes/tokens with the guarantee that: for any k, if k tokens enter the tree then when all the k tokens exit, the tokens will receive sequential numbers, i.e. numbers in the sequence $1, \ldots, k$, without "omissions" or "duplicates". Furthermore, a dynamic tree construction must include a procedure for making *reactive decisions*—i.e. for deciding the tree size that is appropriate for the degree of contention—and a procedure to do the corresponding *adjustments*. The adjustments consist of (i) modifying the size of the tree, (ii) assigning proper $init(C)$ values to the new counters/leaves in the adjusted tree, based on the reactive decisions, and (iii) providing the tokens that traverse the modified parts of the tree with a guarantee about the correctness of the received values.

The correctness of the returned values as well as of the procedure to do adjustments implies the need for proper coordination among processes. For the ST-tree construction we design a data structure that allows for efficient adjustments, and for the coordination we employ fine-grained locking (e.g. to protect access the leaves/counters), as well as *lock-free synchronization* procedures (e.g. to traverse the balancers). In the following Section 2.2 we elaborate on the terms related with the synchronization, while the algorithm itself is given in Section 3.

The requirement for a reaction–decision procedure involves resolving a problem where, if we had some information about the future, we could find an optimal solution, but it is impossible to obtain that kind of information. Such problems can be addressed with on-line algorithms and this is also the approach that we took in the ST-tree construction. In Section 2.3 we elaborate on the terms related with online methods, while our online reaction–decision making algorithm is explained in Section 4.

[1] The terms "performance" and "throughput" are used interchangeably.

$\mathbf{TAS}(x)$

$atomically\{\ oldx := x;\ x := 1;\ \mathbf{return}\ oldx;\ \}$ /* *init:* $x := 0$ */

$\mathbf{CAS}(x, old, new)$

$atomically\{\ oldx := x;\ \mathbf{if}(x = old)\ \mathbf{then}\ \{x := new\};\ \mathbf{return}\ oldx;\ \}$

Fig. 2. The TAS and CAS synchronization primitives.

2.2. Lock-free synchronization

Lock-free methods allow multiple processes (or processors or threads) to access shared data at the same time, but without enforcing mutual exclusion [14,6]. They guarantee that regardless of the contention caused by concurrent operations and the interleaving of their steps, at each point in time there is at least one operation which is able to make progress. This is typically achieved with the use of fine-grained synchronization, with attempts to verify non-interfered access to small amounts of shared data and/or to commit updates using certain common synchronization primitives, like *test-and-set* and *compare-and-swap* (TAS, CAS; cf. Fig. 2, where x is a variable and old, new are values). If such an attempt fails, then that process/thread needs to *retry*. The above may happen due to preemption or due to interleaving by other threads/processes. A method proposed to reduce the fail-retry overhead is the *helping*-method: an operation that detects that it has interleaved steps with another operation, can help the latter operation to progress before proceeding with its own steps, so as to reduce the fail-retry overhead.

The system model considered in the paper is the shared-memory multiprocessor system. Processors may concurrently lead tokens via the tree, in which, as mentioned earlier in this section, balancers are records, while wires are pointers from one record to another, having tokens (processors) visiting them asynchronously. In each step a processor may read or write an atomic variable or execute a CAS or TAS atomic operation. No assumption on the relative speed between processors is made. The synchronization primitives used in the ST-tree construction, are comparable with those used in the original reactive diffracting tree, which are *swap*, TAS and CAS [8].

Linearizability [16] is a property that guarantees that although access to the shared data by different processes/processors takes place over a period of time and may be interleaved, the effect is the same as if each process accesses the data in an atomic time-instance and is sequential relative to the others.

2.3. Online algorithms

Online problems are optimization problems where both the input is received online and the output is produced online with the goal of minimizing the cost of processing the input or maximizing the outcome/profit. If we know the whole input in advance, we may find an *optimal offline algorithm* processing the

whole input with minimal cost or with maximal profit. In order to evaluate how good an online algorithm is, the concept of *competitive ratio* has been suggested.

Competitive ratio: An online algorithm *ALG* is considered competitive with a competitive ratio c (or *c-competitive*) if there exists a constant α so that for any finite input I [26]:

$$OPT(I) \leqslant c \cdot ALG(I) + \alpha, \tag{1}$$

where $ALG(I)$ and $OPT(I)$ are the *profits* of the online algorithm *ALG* and the optimal offline algorithm *OPT* on the input I, respectively. We focus on the profit-centric aspect, as this is the one of relevance to the goal of the reactive trees. (Note that smaller competitive ratio implies better performance for the online algorithm.)

A common way to analyze an online algorithm is to consider a game between an *online player* and a malicious *adversary*. In this game, (i) the online player applies the online algorithm on the input generated by the adversary and (ii) the adversary, with the knowledge of the online algorithm, tries to generate the worst possible input to maximize the competitive ratio c.

For distributed algorithms, besides the unpredictable input, there exists another source of non-determinism, namely the *time uncertainty* (or *schedule*), that includes interleaving of process steps [2,15]. It is possible to analyze against a powerful adversary who has knowledge of the global state of the system each time, but it has also been shown that in concurrent systems, a more precise characterization of the competitiveness can be reached when the analysis is done using an adversary who does not have this power [2].

Regarding the reactive decision making in the ST-trees, the point of interest is how good a profit is achieved by the decisions of the online player, namely the ST-tree reactive-decision-making procedure associated with each leaf. The analysis considers that the player has to make a decision based on the current and previous values of the contention at that leaf and it is "opposed" by an adversary who decides on the future input, i.e. adapts the contention in the future steps in a way that will be adverse to the leaf's decision and incur a larger competitive ratio. The analysis refers to each leaf separately. We elaborate on the power of the adversary and why the above is a justified case, in Section 4, in connection with the presentation and analysis of the reaction-decision method for the ST-trees.

3. The ST-tree construction

In this section we focus on the data structure designed for the ST-trees and the coordination algorithm of the construction, namely the traversal procedure and the procedure to do the adjustments instructed by reactive decisions. The algorithm to make the reactive decisions is detailed in Section 4.

3.1. The tree structure

To adapt to contention variation efficiently, each leaf in the tree should be able to shrink and grow freely to any level suggested by the reactive decision and the adjustment should

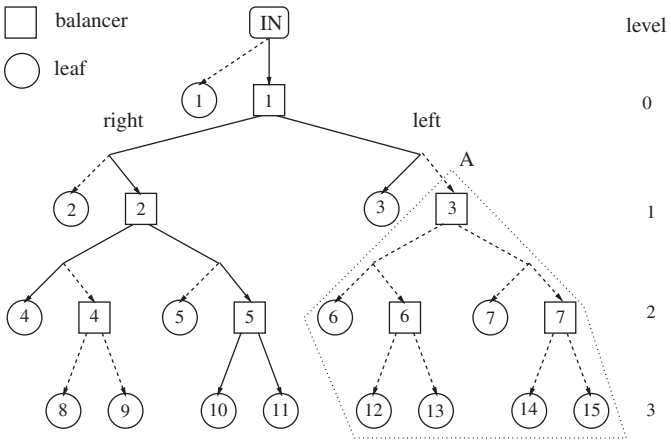

Fig. 3. A self-tuning reactive tree.

be able to complete in as small number of steps as possible. These are so because of efficiency reasons, as well as because the contention situation may change *during* an adjustment, causing conflicting adjustments to start. A complication during adjustments is due to the processors (tokens)[2] concurrently traversing the parts of the tree that are being modified. These processors have to be taken into account in setting the initial values of the counters in the new subtrees. In [8] the problem is addressed by having these processors go back to higher levels of the tree and start traversing again. This, under unfortunate scenarios, may result in an unbounded number of traversals for these processors.

This motivated us to investigate the possibility of a different solution, which minimizes the adjustment (hence, obstruction) time. A basic component of this solution is a new data structure for the tree that uses other types of fine-grain synchronization to take these processors into account when setting the initial values of the counters.

Fig. 3 illustrates the data structure of the self-tuning tree. The squares in the figure are balancers and the circles are leaves. The numbers in the squares and circles are their labels. Each balancer has a *matching* leaf with the same label number. Symmetrically, each leaf that is not at the lowest level of the tree has a *matching* balancer. Each balancer has two outputs, *left* and *right*, each of which is a pointer that can point to either a leaf or a balancer. A *Shrink* or *Grow* operation is essentially a switch of such a pointer (from a balancer to the matching leaf or vice versa). Solid arrows in the figure represent the current pointer directions.

In a system with n processors, the algorithm needs $n-1$ balancer nodes and $2n-1$ leaf nodes. Note that although the ST-tree introduces an auxiliary node (matching leaf) for each balancer, this does not require extra space compared with the original reactive diffracting tree (RD-tree). This is because the terms refer to virtual nodes, introduced only to split the *functionality* of each node in the RD-tree into two "roles" (or virtual nodes) in the ST-tree: one virtual node is enabled when

[2] As each of the tokens traversing the network concurrently is led by a processor, the terms "token" and "processor" are used interchangeably.

the node plays the role of a balancer and the other virtual node is enabled when the node plays the role of a leaf. The interplay and synchronization algorithm of "old" and "new" tokens (relative to the point when a node switches from a balancer to a leaf or vice versa) is explained with the algorithms of the expanding and shrinking adjustments in Sections 3.3 and 3.4, while cf. Sections 5.2 and 5.3 also provide pseudocode and implementation-level details.

3.2. The traversal procedure

To traverse the tree, a processor p_i first visits the tree at its root *IN* and then follows the root pointer to visit balancer 1. When visiting a balancer, p_i switches the balancer's toggle-bit to the other position (i.e. from left to right and vice versa) and then follows the current direction of the corresponding pointer to visit the next node. Here it is possible to use the elimination method [24], which has the potential to evenly balance each pair of incoming tokens (and direct them left and right) without accessing the toggle-bit. When p_i visits a leaf L, before taking an appropriate counting value and exiting, it executes a procedure that estimates which level is the best for L with the current contention. The outcome can imply that the leaf needs to grow into a subtree or to shrink (together with siblings in a subtree) or to stay as it is. An elaborate description of the algorithm to make the reactive decisions is deferred to Section 4.

3.3. The adjustment procedure: expanding a leaf into a subtree

If the recommended action resulting from the reactive scheme is to grow to a level l_{lower}, i.e. the current contention on the leaf L is too high and L should grow to the level l_{lower}, the processor p_i visiting the leaf must help L carry out the growth task before exiting the tree. The task consists of (i) constructing the corresponding subtree (if it did not already exist), (ii) assigning initial values to the counters of the new subtree, and (iii) switching the corresponding pointer from L to its matching balancer, which is the root of the subtree. For instance, assume that p_i is visiting leaf 3 in Fig. 3 and the recommendation is to grow to a subtree A with a depth of 3; p_i first constructs the subtree (while other processors may normally access leaf 3 and exit the tree without any disturbance) and then locks leaf 3 in order to (i) switch the pointer to balancer 3 and (ii) assign *proper* values to counters 12, 13, 14 and 15; then it releases leaf 3. At this point, the incoming processors following the left pointer of balancer 1 traverse the new subtree while the old processors that were already directed to leaf 3 continue to access leaf 3 and exit the tree. After completing the growth task, p_i increases the counting value of leaf 3 and exits the tree.

To assign proper initial values to the counters of the new subtree, the *Grow* procedure needs to take care of the tokens that are already waiting to traverse L itself, and the (old) tokens that may still be traversing the (new) subtree. Moreover, it needs to coordinate with possible concurrent *Shrink* procedure(s) that might have been initiated by some higher-level balancer(s) and with possible concurrent *Grow* procedures. In particular L gets locked by the *Grow* procedure (to ensure a *closed* set of processors contending for it). Similarly, the leaves of the new subtree get locked, in order to get assigned with new initial values. Note that all this locking is *conditional*, i.e. if a lock-attempt fails, the adjustment procedure aborts, as the failure to acquire the lock implies that a competing adjustment procedure is ongoing and is "winning". (Complementary actions are taken by the *Shrink* procedure; cf. description below.) The exact synchronization order is given in Section 5.2, together with the pseudocode.

3.4. The adjustment procedure: shrinking a subtree into a leaf

If the recommended action is to shrink to a level l_{higher}, i.e. the current contention on the leaf L is too low and L should shrink to the higher level l_{higher} so as to reduce the travel latency, the pointer to the ancestral balancer of L at level l_{higher}—let it be denoted by B—must switch to the matching leaf whose counting value must be set properly. Since the subtree rooted at B contains other leaves that may have not recommended to shrink to l_{higher}, a method to coordinate them is needed. We propose an asynchronous vote-collecting scheme for this purpose. The leaf L *votes* for the level l_{higher} by adding its *weighted vote* to B's "vote box".

Definition 1. The weight of a leaf's vote is the number of leaves at the lowest level in the subtree rooted at the matching balancer.

For instance, in Fig. 3 the weight of leaf 4's vote is 2 since the vote represents two leaves at the lowest level, namely 8 and 9.

After voting, the leaf (i.e. the processor carrying out the task on its behalf) checks whether there is a majority in B's subtree that has voted similarly (The leaf can require all the leaves, instead of a majority, in B's subtree to vote if the contention on the minority is the main concern.). If so, the *Shrink* procedure is initiated. Similarly as the *Grow* procedure, the *Shrink* procedure locks the matching leaf and the leaves of the subtree B in order to (i) collect their counting values, (ii) compute the *proper* counting value for the new matching leaf and (iii) switch the pointer from B to its matching leaf. Note that all the leaves of the subtree B need to be locked *only if* the contention on the subtree is *so low that the subtree should shrink to a leaf*. Therefore, locking the subtree in this case affects performance as much as locking a leaf in the original reactive diffracting tree.

As an example, assume that a processor p_i visits leaf 10 in Fig. 3 and the recommendation is to shrink to level 1. Assume that leaf 4 has voted for balancer 2 too. The weight of leaf 4's vote is two since the vote represents leaves 8 and 9 at the lowest level. Leaf 10's vote has a weight of 1. Therefore, the sum of the vote weights at balancer 2 is 3. In this case, p_i helps balancer 2 execute the shrinkage task since three of four leaves at the lowest level, leaves 8, 9 and 10, have voted for the balancer. Subsequently, (i) p_i locks leaf 2 and all the leaves of the subtree rooted at balancer 2, (ii) collects their counting

values, (iii) computes the next counting value for leaf 2 and (iv) finally switches the pointer from balancer 2 to leaf 2. After that, all the leaves of the subtree are released immediately so that other processors can continue to access them. As soon as leaf 2 is assigned the new value, new processors going along the right pointer of balancer 1 access the new leaf and exit the tree while old processors continue traversing the old subtree. After completing the shrinkage task, p_i increases the counting value of leaf 10 and exits the tree.

Both the growing and shrinking processes support high parallelism: incoming processors follow the new subtree/leaf while pending processors continue traversing the old leaf/subtree.

4. Online method for making reactive decisions in the ST-tree

4.1. Modeling the reactive-decision problem

In order to be self-tuning, each leaf of the new tree needs to be able to locally decide whether to shrink or grow depending on its current contention, without external help (e.g. without hand-tuned parameters).

As we have seen in Section 2.1, in order to maximize throughput the tree size must be appropriate for the current contention level, minimizing both the tokens' *latency due to the contention* at the leaves and their *latency due to traversal* from the root to the leaves. However, these two latency-related factors are opposed to each other, i.e. if we want to decrease the contention at the leaf, we need to move the leaf to a lower level and thus the traversal costs increase.

In this section, we model the problem of balancing the trade-off between the two latency-related factors as an online problem and subsequently suggest a solution, which is inspired by the *threat-based policy* [10].

4.1.1. Estimating the contention

Let P denote the maximum number of processors that can access the tree simultaneously. Recall from Section 2.1 that P is assumed known. The assumption is feasible since P can be chosen as the number of processors in the system.

Each leaf L locally estimates the contention of the tree T via its own, local contention:

$$contention_T = contention_L * 2^{\text{level}_L}, \tag{2}$$

where $contention_L$ is measured by the number of processors that are trying to access L simultaneously and $level_L$ is the level of L. When the tree T degenerates to a single leaf or, in other words, L is at the highest level (i.e. level 0), the tree's contention is the leaf's contention. At the beginning, the tree degenerates to a single leaf, the root leaf, which considers expansion to a tree only if there is contention between at least two processors. Therefore,

$$2 \leqslant contention_T \leqslant P, \tag{3}$$

where $contention_T$ varies unpredictably.

4.1.2. Estimating the appropriate (sub)tree depth

Since the ideal contention on each leaf occurs when only one processor is accessing a leaf at a time, the number of leaves need not be more than P. Let *surplus* denote the subtraction of the number of leaves from P. The tree adjusts the *surplus* (or changes its size) to control the contention on the leaves. However, since the tree is adjusted by the local decision at each leaf, each leaf L locally estimates the *surplus*:

$$surplus_L = P - 2^{\text{level}_L}, \tag{4}$$

and then adjusts its level to change the estimated *surplus*. We have lower/upper bounds of $surplus_L$: $0 \leqslant surplus_L \leqslant P - 1$.

Let $tr\_latency$ denote the latency due to token/processor's traversal from the root to the leaf L. Intuitively, when L grows or moves to a lower level, $tr\_latency$ will increase since tokens must traverse additional intermediate levels. However, the increased amount of $tr\_latency$, $\Delta tr\_latency$, depends on a complex combination of many factors: the memory-access latency from the tokens/processors to balancers at additional intermediate levels and the access delays due to the contention on the balancers. Therefore, the relation between $\Delta tr\_latency$ and $\Delta surplus$ (the increase in size of the tree) is complicated for theoretical analysis. We simplify the relation by the following observation. When the tree increases its size by $\Delta surplus$ at its current contention $contention_T$, the product $\Delta surplus \cdot contention_T$ is a $tr\_latency$-related value. The value can be accumulated as $tr\_latency$-savings that can be converted back to *surplus* to reduce the tree's size when the tree's contention $contention_T$ decreases. Since the tree's growth is decided by its leaves, each leaf L locally accumulates its $tr\_latency$-savings:

$$tr\_latency_L = \sum_{adjustments} \Delta surplus_L \cdot contention_T, \tag{5}$$

where $tr\_latency_L$ is 0 when the tree initially consists of only the root leaf.

Definition 2. A *contention-rising* (*resp. contention-dropping*) *phase* for a leaf L is a maximal sequence of subsequent visits at the leaf with a monotonic non-decreasing (resp. non-increasing) locally estimated $contention_T$. A contention-rising phase ends when a decrease in contention is observed; at that point a contention-dropping phase begins.

4.1.3. Contention-rising phases

Let us consider a contention-rising phase for a leaf L. The challenge is that the leaf must locally decide the new *appropriate* level to grow when observing an increase of $contention_T$. If L grows too much, it generates an unnecessarily long traversal latency for tokens. If L grows too little, it causes long access latency for tokens at the leaves due to their contention. Therefore, L needs to find the *appropriate* level, i.e. the level that can minimize the sum of these latencies for the tokens coming from the root.

We model the problem as an online problem in which the online player, representing the leaf L, is seeking an optimal level for each contention increase observed in a contention-rising

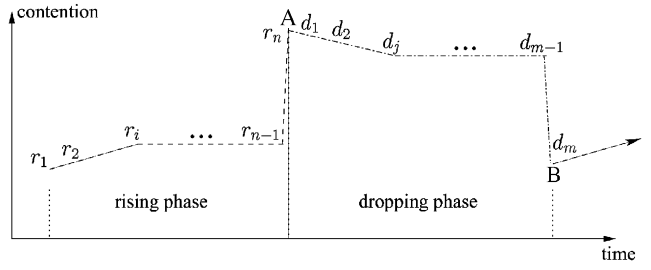

Fig. 4. An illustration for the adversary's strategy.

phase. The sequence of contention increases unfolds sequentially. When observing an increase, without knowledge of the future contention variation, the player must decide how much the leaf L should grow. On the other hand, the malicious adversary, with knowledge of the player's algorithm, generates the worst sequence of contention increases for the player, trying to make the player expand the leaf inappropriately. Here we are interested in designing a deterministic online algorithm. Randomized online algorithms for the problem can be a topic of future research.

Fig. 4 illustrates how the adversary could fool the player. Since the tree initially consists of only the root leaf, we have $surplus_L = P - 1$ and $tr\_latency_L = 0$ at the beginning. Assume that the adversary knows A as the optimal contention-point for the leaf L to grow completely (i.e. $surplus_L = 0$) and B as the optimal contention-point for L to shrink completely (i.e. $tr\_latency_L = 0$). Since the malicious adversary knows the deterministic algorithm used by the player, she can generate a sequence of contention increases $r_1 \leqslant r_2 \leqslant \cdots \leqslant r_i \leqslant \cdots \leqslant r_n = A$ that may fool the player to either never expand L or expand L to the maximum level too early. Assume that when the player observes a contention increase, she expands the leaf L according to the deterministic algorithm. The leaf eventually reaches its maximum level (i.e. $surplus_L = 0$) at a low-contention level, for instance, r_i before the contention reaches the point A. The adversary keeps the low-contention level until the end of the contention-rising phase at which she immediately increases the contention level to A. Consequently, the player may cause an *unnecessarily* long traversal latency for tokens coming from the root to L during the long interval $[r_i, r_{n-1}]$.

The *goal* of an algorithm for the problem of deciding how much to grow is to *maximize* the following function:

$$\mathcal{F} = \sum_{l \in T_k} \Delta surplus_{L,l} \cdot r_l, \tag{6}$$

for each contention-rising phase T_k, where $\Delta surplus_{L,l}$ is the amount of $surplus_L$ that the player "spends" at contention r_l to expand the leaf L. For the malicious sequence of contention increases in Fig. 4, the player achieves a small $\mathcal{F}$ since she spends all her surplus "budget" $P - 1$ at low-contention levels $r_1 \leqslant r_2 \leqslant \cdots \leqslant r_i$. The adversary, however, spends her surplus "budget" at the highest contention-point A in the contention-rising phase, achieving the largest $\mathcal{F}$.

4.1.4. Contention-dropping phases

Similarly, for the contention-dropping phase from A to B the adversary can generate a sequence of contention decreases $d_1 \geqslant d_2 \geqslant \cdots \geqslant d_j \geqslant \cdots \geqslant d_m = B$ that may fool the player to either never shrink the leaf L or shrink the leaf L to the minimum level too early. Assume that when the player observes a contention decrease, she shrinks the leaf L according to the deterministic algorithm. The leaf eventually may reach its minimum level (e.g. the root leaf) at a high-contention level, for instance, d_j before the contention reaches the point B. Consequently, the player may cause a high-contention on the leaf L, generating a long access latency for L's incoming tokens during the long interval $[d_j, d_{m-1}]$. The goal of an algorithm for the problem of deciding how much to shrink is to maximize the following function:

$$\mathcal{F} = \sum_{l \in T_k} \Delta tr\_latency_{L,l} \cdot \frac{1}{d_l} \tag{7}$$

for each contention-dropping phase T_k, where $\Delta tr\_latency_{L,l}$ is the amount of $tr\_latency_L$ that the player needs to convert back to $surplus_L$ at contention d_l in order to shrink the leaf L to a higher level.

4.1.5. The goal of the decision-making algorithm

In summary, the problem of finding an appropriate level for a leaf L at a certain contention level $contention_T$ can be modeled as an online problem. The sequence of contention values $contention_T$ unfolds sequentially. When observing a new contention value, the online player, with knowledge of the contention lower/upper bounds $2 \leqslant contention_T \leqslant P$, must decide how much the leaf L should grow/shrink with respect to the current contention by converting either a fraction of $surplus_L$ to $tr\_latency_L$ (for growth) or a fraction of $tr\_latency_L$ to $surplus_L$ (for shrinkage). The goal is to maximize

$$\mathcal{F} = \sum_{l \in T_k} \Delta surplus_{L,l} \cdot r_l \tag{8}$$

for each contention-rising phase T_k: $r_1 \leqslant r_2 \leqslant \cdots \leqslant r_n$, or to maximize

$$\mathcal{F} = \sum_{l \in T_k} \Delta tr\_latency_{L,l} \cdot \frac{1}{d_l} \tag{9}$$

for each contention-dropping phase T_k: $d_1 \geqslant d_2 \geqslant \cdots \geqslant d_m$.

4.1.6. The role of the adversary

In the algorithm and the analysis we treat the game between the adversary and each player/leaf independently from the game of the adversary with the other leaves. This is because the adversary, with its power to schedule the tokens (processors), i.e. to generate new traversal requests and to propagate them independently across each link with arbitrary speed, can adjust the level of contention at each leaf, independently from the levels of contention at the other leaves. Low contention can be achieved by the adversary by moving the tokens slowly

toward the leaves. High contention is achievable by the adversary by inserting new tokens and scheduling them fast towards the preferred leaves. From the adversary's point of view, this is like a resource allocation problem, where the number of resources (the maximum number of concurrent tokens P) is known and processes (leaves) place demands on the number of resources with a known maximum. The demands are contention levels desirable by the adversary at each leaf in order to fool the corresponding player. By acting using the Banker's algorithm [9], the adversary is able to compete with each player independently.

4.2. The decision-making algorithm

Our algorithm for the online problem of reactively adjusting a leaf L is inspired by the *threat-based policy*, presented in [10].

Let us first consider a contention-rising phase (a contention-dropping phase is symmetric). The idea of our algorithm is to expand the leaf L *just enough* to guarantee a bounded competitive ratio (with respect to the outcome $\mathcal{F}$) even if the contention suddenly drops to the minimum in the next observation. In particular, the algorithm follows the rules:

R1. The leaf L grows only when the contention $contention_T$ is the highest so far in the present contention-rising phase.

R2. (*Considering the threat*). When growing, it grows *just enough* to guarantee a competitive ratio c^* even if the contention drops to the minimum in the next observation.

Below follows an intuitive description of the exact decisions, together with a more formal analysis of the formulas, following the analysis of the *threat-based policy* [10].

Recall that we are considering a contention-rising phase with observed levels of contention $r_1 \leqslant r_2 \leqslant \cdots \leqslant r_i \ldots$. At the moment, assume that the online player knows a competitive ratio c^* that is achievable by a (deterministic) online algorithm. We will present below how the player can calculate c^* using P, the maximum number of processors that can access the tree simultaneously. Note that $c^* = \frac{\mathcal{F}_{\text{Adversary}}}{\mathcal{F}_{\text{Player}}}$ where $\mathcal{F}_{\text{Adversary}}$ and $\mathcal{F}_{\text{Player}}$ are the outcomes $\mathcal{F}$ (cf. Eq. (6)) of the adversary and the player, respectively. At the first contention observed, r_1, the player must expand the leaf L by converting an amount s_1 of $surplus_L$ such that the competitive ratio c^* is guaranteed even if the contention drops to the minimum in the remaining steps. Since c^* is achievable by a deterministic online algorithm, there exists such an s_1 value. Choosing such s_1 the player can ignore contention values not greater than r_1 in the remaining steps while still guaranteeing the competitive ratio c^*. This results in the first rule mentioned above (R1).

On the other hand, if s_1 is always chosen as much as the whole $surplus_L$ budget, the player will not be able to expand the leaf L further when the contention continues increasing. This consequently may cause the player being unable to guarantee the competitive ratio c^*. Therefore, s_1 should be the *minimum* amount that guarantees the competitive ratio c^*. This argument results in the second rule mentioned above (R2). The amounts s_i to be converted in the rest of the phase can be inductively justified using similar arguments.

Lemma 3. *In a contention-rising phase, the formula for calculating the amount of* $surplus_L$ *to be converted to* $tr\_latency_L$ *when the contention observed is* $contention_T$ *and the highest preceding contention observation in the phase is* $contention_T^-$, *is*

$$\Delta surplus_L = baseSurplus_L \cdot \frac{1}{c^*} \cdot \frac{contention_T - contention_T^-}{contention_T - 2}, \tag{10}$$

where $baseSurplus_L$ *is* $surplus_L$ *at the beginning of the phase and* c^* *is a known competitive ratio that is achievable by a* (*deterministic*) *online algorithm. The new* (*lower*) *level of the leaf* L *is calculated using its remaining* $surplus_L$: $level_L = \log_2(P - surplus_L)$.

Proof. Because of the first rule, the player considers only the contention sequence of successive maxima $2 \leqslant m_1 < m_2 < \cdots \leqslant P$ (cf. Eq. (3)) in the sequence of observed contention levels $r_1 \leqslant r_2 \leqslant \cdots \leqslant r_i \leqslant \cdots \leqslant r_n$. Consider a step i, in which a new maximum, m_i is observed. Let R_i and A_i be the amount of remaining $surplus_L$ and the amount of accumulated $tr\_latency_L$ right after the ith step, respectively. (At the beginning of the phase, $A_0 = 0$ and $R_0 = baseSurplus_L$.) From rule R2, we have

$$\frac{R_0.m_i}{R_i.2 + A_i} \leqslant c^*, \tag{11}$$

where the denominator $(R_i.2 + A_i)$ is the player's outcome in the case that the contention drops to the minimum (i.e. to 2), right after the ith step and stays there. Consequently, the remaining surplus R_i is calculated as it is converted at the minimum contention. The numerator is the adversary's outcome in this case: she converts her surplus "budget" R_0 at the highest contention m_i in the sequence of contention increases created by her.

If $s_i = R_{i-1} - R_i$ denotes the amount of $surplus_L$ converted to $tr\_latency_L$ at step i, Eq. (11) becomes

$$\frac{R_0.m_i}{2(R_{i-1} - s_i) + (A_{i-1} + s_i.m_i)} \leqslant c^*. \tag{12}$$

Since function $\mathcal{G}(s_i) = \frac{R_0.m_i}{2(R_{i-1}-s_i)+(A_{i-1}+s_i.m_i)}$ is decreasing with s_i due to $m_i \geqslant 2$, the minimum value for s_i is found when $\mathcal{G}(s_i) = c^*$, i.e.

$$\frac{R_0.m_i}{2R_i + A_i} = \frac{R_0.m_i}{2(R_{i-1} - s_i) + (A_{i-1} + s_i.m_i)} = c^*. \tag{13}$$

It follows that

$$s_i = \frac{1}{m_i - 2}\left(\frac{R_0.m_i}{c^*} - (2R_{i-1} + A_{i-1})\right). \tag{14}$$

For the case $i = 1$:

$$s_1 = \frac{1}{m_1 - 2}\left(\frac{R_0.m_1}{c^*} - (2R_0 + A_0) =\right) = \frac{R_0}{c^*} \cdot \frac{m_1 - 2c^*}{m_1 - 2}. \quad (15)$$

For the case $i \geqslant 2$, it holds that $2R_{i-1} + A_{i-1} = \frac{R_0.m_{i-1}}{c^*}$ (Eq. (13)). Hence Eq. (14) becomes

$$s_i = \frac{1}{m_i - 2}\left(\frac{R_0.m_i}{c^*} - \frac{R_0.m_{i-1}}{c^*}\right) = \frac{R_0}{c^*} \cdot \frac{m_i - m_{i-1}}{m_i - 2}. \quad \square \quad (16)$$

The following shows how the player calculates the competitive ratio c^* using the available information.

Since the length of the contention sequence of successive maxima in the contention-rising phase is unknown to the player, she must consider an arbitrarily long contention sequence, i.e. a contention sequence with length $k \to \infty$. From the definition of s_i, we have

$$\lim_{k\to\infty} \sum_{i=1}^{k} s_i = \lim_{k\to\infty} (R_0 - R_k) \leqslant R_0. \quad (17)$$

The player should leave no remaining surplus at the end of the phase (i.e. $R_k = 0$) since the remaining amount R_k must be converted to $tr\_latency_L$ at the minimum exchange rate, which makes the player's outcome worse. This results in

$$\lim_{k\to\infty} \sum_{i=1}^{k} s_i = R_0$$

$$\Leftrightarrow \frac{R_0}{c^*}\left(\frac{m_1 - 2c^*}{m_1 - 2} + \lim_{k\to\infty} \sum_{i=2}^{k} \frac{m_i - m_{i-1}}{m_i - 2}\right) = R_0$$

$$\Leftrightarrow c^* = 1 + \frac{m_1 - 2}{m_1} \lim_{k\to\infty} \sum_{i=2}^{k} \frac{m_i - m_{i-1}}{m_i - 2}.$$

Since the malicious adversary generates the contention sequence that makes c^* as large as possible, this implies

$$c^* = 1 + \max_{2 \leqslant m_1 < m_2 < \cdots \leqslant P} \left(\frac{m_1 - 2}{m_1} \lim_{k\to\infty} \sum_{i=2}^{k} \frac{m_i - m_{i-1}}{m_i - 2}\right). \quad (18)$$

We can prove that $c^* \leqslant \varphi - \frac{\varphi - 1}{\varphi^{1/(\varphi-1)}}$, where $\varphi = \frac{P}{2}$, which implies that the online algorithm obeying the aforementioned rules with $c = \varphi - \frac{\varphi-1}{\varphi^{1/(\varphi-1)}}$ will achieve the competitive ratio c. The proof is rather tedious, following the proof in [10], to which the interested reader is referred for the details.

For a contention-dropping phase, the player can use a symmetric algorithm to shrink the leaf L to a higher level by converting the savings $tr\_latency_L$ back to $surplus_L$. This results in the following lemma:

Lemma 4. *In each contention-rising/contention-dropping phase, the reactive adjustment method of ST-trees is competitive with competitive ratio* $c = \varphi - \frac{\varphi-1}{\varphi^{1/(\varphi-1)}} = \Theta(\ln \varphi)$, *where* $\varphi = \frac{P}{2}$ *and P is the maximum number of processors that can access the tree simultaneously.*

5. Implementation of the ST-trees

Fig. 5 describes the basic data structure and shared variables used in the tree implementation.

5.1. Traversing self-tuning reactive trees

A processor *Pid* traverses the tree by calling the *TraverseTree*() function in Fig. 6. First, it visits *Balancers*[0] (the "IN" node in Fig. 3), whose left child points to either *Balancers*[1] or *Leaves*[1]. Before visiting a node on the next level, it updates its new location in *Tracing*[*Pid*] (line T0). It records to its private variable *MyPath* the path along which it traverses the tree, where *MyPath*[*i*] is the node visited at level *i*. At each node, its behavior depends on the node type.

If the node is a balancer, the processor calls the *TraverseB* procedure to follow an appropriate child link (line T2 in *TraverseTree*()) and subsequently updates its new location to *Tracing*[*pid*] (lines B1, B2 in *TraverseB*()). In the *TraverseB*

```
type NodeType = record Nid : [1..MaxNodeId]; kind : {BALANCER, LEAF}; mask: boolean; end; /*in one word*/
     BalancerType = record state : {ACTIVE, OLD}; level : int; toggleBit : boolean; left, right : NodeType; end;
     LeafType = record state : {ACTIVE, OLD}; transPhase : {RISING, DROPPING}; count, init : int;
                level, oldLevel, newLevel, contention, load, surplus, latency : int; end;
shared variables
     Balancers : array[0..MaxNodeId] of BalancerType; Leaves : array[1..MaxNodeId] of LeafType;
     TokenToReact : array[1..MaxNodeId] of boolean; Tracing : array[1..P] of [1..MaxNodeId];
```

Fig. 5. The basic data structure for the ST-tree.

```
int TraverseTree( int Pid)
T0  n := Assign(&Tracing[Pid], &Balancers[0].left);
    for(i := 1;; i++) do
T1    MyPath[i] := n;
T2    if IsBalancer(n) then
        n := TraverseB(Balancers[n.Nid], Pid)
      else /*IsLeaf*/
T3      Leaves[n.Nid].contention++;
T4      if (Leaves[n.Nid].state = ACTIVE) and
          (TAS(TokenToReact[n.Nid]) = 0) then
          react := CheckCondition(Leaves[n.Nid]);
T5        if react = SHRINK then
            Elect2Shrink(n.Nid, MyPath);
T6        else if react = GROW then
            Grow(n.Nid);
T7        Reset(TokenToReact[n.Nid]);
T8      result := TraverseL(n.Nid);
T9      Leaves[n.Nid].contention--;
T10     Assign(&Tracing[Pid], &Balancers[0]);
        /*reset Tracing[Pid]*/
T11     return result;

NodeType TraverseB(BalancerType B, intPid)
B0  if ((k := Toggle(B.toggleBit)) = 0) then
B1    return (Assign(&Tracing[Pid], &B.right));
B2  else
      return (Assign(&Tracing[Pid], &B.left));

int TraverseL( int Nid)
L0  L := Leaves[Nid];
L1  AcquireLock(L.lock, Nid); /*lock the leaf*/
L2  result := L.count;
    L.count := L.count + 2^L.level;
L3  Release(L.lock); /*release the leaf*/
L4  return result;

int CheckCondition(LeafType L)
C0  Load := MIN(P, L.contention * 2^L.level);
C1  FirstInPhase := False;
C2  if (L.transPhase = RISING) and
        (Load < L.load) then
        L.transPhase := DROPPING;
        FirstInPhase := True;
C3  else if (L.transPhase = DROPPING) and
        (Load > L.load) then
        L.transPhase := RISING;
        FirstInPhase := True;
C4  if L.transPhase = RISING then
        Surplus2TrLatency(L, Load, FirstInPhase);
C5  else
        TrLatency2Surplus(L, 1/Load, FirstInPhase);
C6  L.newLevel := log2(P − L.surplus);
C7  if L.newLevel < L.level then return SHRINK;
C8  else if L.newLevel > L.level then return GROW;
C9  else return NONE;

Surplus2TrLatency(L, Load, FirstInPhase)
SL0 X := L.surplus; baseX := L.baseSurplus;
    Y := L.latency;
SL1 rXY := Load; LrXY := L.load;
SL2 if FirstInPhase then
      if rXY > mXY * C then
      /* mXY: lower bound of rXY, C: comp. ratio*/
        ΔX := baseX * 1/C * (rXY − mXY*C)/(rXY − mXY);
SL3 else
      ΔX := baseX * 1/C * (rXY − LrXY)/(rXY − mXY);
SL4 L.surplus := L.surplus − ΔX;
SL5 L.latency := L.latency + ΔX * rXY;

TrLatency2Surplus(L, 1/Load, FirstInPhase)
/* symmetric to the above with: X := L.latency;
Y := L.surplus; */
```

Fig. 6. The TraverseTree, TraverseB, TraverseL, CheckCondition, Surplus2TrLatency and TrLatency2Surplus procedures.

procedure, the toggle mechanism for toggle-bits can be implemented using either advanced techniques like the elimination technique [24] to alleviate the contention on toggle-bits or low-contention hardware primitives like the *fetchop* primitives in the SGI Origin2000 [18].

If the node is a leaf, in all cases the processor calls the *TraverseL* procedure to read and increase the leaf counter $L.count$ (line T8) and resets *Tracing*[*pid*] to the tree root (line T10). If exiting the tree through a leaf whose state is *ACTIVE*,[3] the processor must actively execute a reactive procedure. It

[3] Meaning that its state is not *OLD*; intuitively, old leaves (and old balancers) are those that a new processor traveling from the root cannot visit at that time.

```
GROW (int Nid) /*Leaves[Nid] becomes OLD;
    Balancers[Nid] and its subtree become ACTIVE*/
G0  L := Leaves[Nid]; B := Balancers[Nid];
G1  forall i, Read(Tracing[i])
      if ∃ pending processors in the subtree B then
        return; /*abort*/
G2  for each balancer B' in the subtree rooted at B,
      B'.toggleBit = 0; /*{B'} includes B*/
G3  for each leaf L' at level L.newLevel of the
      subtree, in decreasing order of nodeId do
      if not AcquireLock_cond(L'.lock, Nid) then
        Release all acquired locks; return; /*abort*/
G4  if (not AcquireLock_cond(L.lock, Nid))
      or (L.state = OLD) then
/*1st: an ancestor activated an overlapping Shrink*/
/*2nd:someone already made the expansion*/
      Release all acquired locks; return; /*abort*/
G5  Switch parent's pointer from L to B;
G6  forall i, Read(Tracing[i])
      ppL := #(pending processors at L);
/*Miss no processor since the new ones go to B*/
G7  CurCount := L.count; L.state := OLD;
G8  Release(L.lock);
G9  for each balancer B' as in step G2 do
      B'.state := ACTIVE;
G10 for each leaf L' as in step G3 do
      L'.count := NextCountG(ppL, CurCount);
      L'.state := ACTIVE;
      Release(L'.lock);
    return;/*Success*/

ELECT2SHRINK( int Nid, NodeType MyPath[])
E0  L := Leaves[Nid];/*the leaf asks to shrink*/
    if L.oldLevel < L.newLevel then
/*new suggested level is lower than older suggestion*/
      for(i := L.oldLevel; i < L.newLevel; i++) do
E1      Balancers[MyPath[i].Nid].votes[Nid] := 0;
    else for(i := L.newLevel; i < L.oldLevel; i++) do
E2      B := Balancers[MyPath[i].Nid];
E3      B.votes[Nid] := 2^(MaxLevel−L.level);
E4      bWeight := 2^(MaxLevel−B.level);
        /*weight of B's subtree*/
E5      if (Σ_j B.votes[j]) / bWeight > 0.5 then Shrink(i); break;

SHRINK ( int Nid)/*Leaves[Nid] becomes ACTIVE;
    Balancers[Nid] and its subtree become OLD*/
S0  B := Balancers[Nid]; L := Leaves[Nid];
S1  forall i : Read(Tracing[i])
      if ∃ pending processor at L then return;/*abort*/
S2  if ( not AcquiredLock_cond(L.lock, Nid))
      or (B.state = OLD) then
      /*1st: some ancestor is performing Shrink*/
      /*2nd: someone already made the shrinkage*/
      Release possibly acquired lock; return; /*abort*/
S3  L.state := OLD; /*avoid reactive adjustment at L*/
S4  forall leaf L' in B's subtree, in increasing order
    of nodeId do
      AcquireLock_cond(L'.lock, Nid);
S5  Switch the parent's pointer from B to L
S6  forall i : Read(Tracing[i])
      eppB := #(effective processors in B's subtree);
      /*can't miss any since the new ones go to L*/
S7  for each balancer B' in the subtree rooted at B do
      B'.state := OLD;/*{B'} includes B*/
    SL := ∅; SLCount := ∅;
S8  for each leaf L' in the subtree rooted at B do
      if (L'.state = ACTIVE) then
        SL := ∪{L'};SLCount := ∪{L'.count};
        L'.state := OLD;
      Release(L'.lock);
S9  L.count := NextCountS(eppB, SL, SLCount);
S10 L.state := ACTIVE;
S11 Release(L.lock);
```

Fig. 7. The *Grow*, *Elect2Shrink* and *Shrink* procedures.

acquires the leaf *TokenToReact* and, if successful, it invokes the *CheckCondition* procedure (line T4). The procedure implements the new reactive scheme mentioned in Section 4. According to the *CheckCondition* procedure result, the processor invokes either the *Grow* procedure to expand the leaf (line T6) or the *Elect2Shrink* procedure to shrink the tree (line T5). The *Grow* and *Elect2Shrink* procedures are presented in Sections 5.2 and 5.3.

5.2. Implementing the grow procedure

An expansion of a leaf L to a subtree T whose root is L's matching balancer B and depth is $L.newLevel - L.level$ essentially needs to set proper counting values for the new leaves in T so as to ensure the counting property. Fig. 8 illustrates the steps taken in the *Grow* procedure whose pseudocode is in Fig. 7. The expansion occurs only if there are no pending

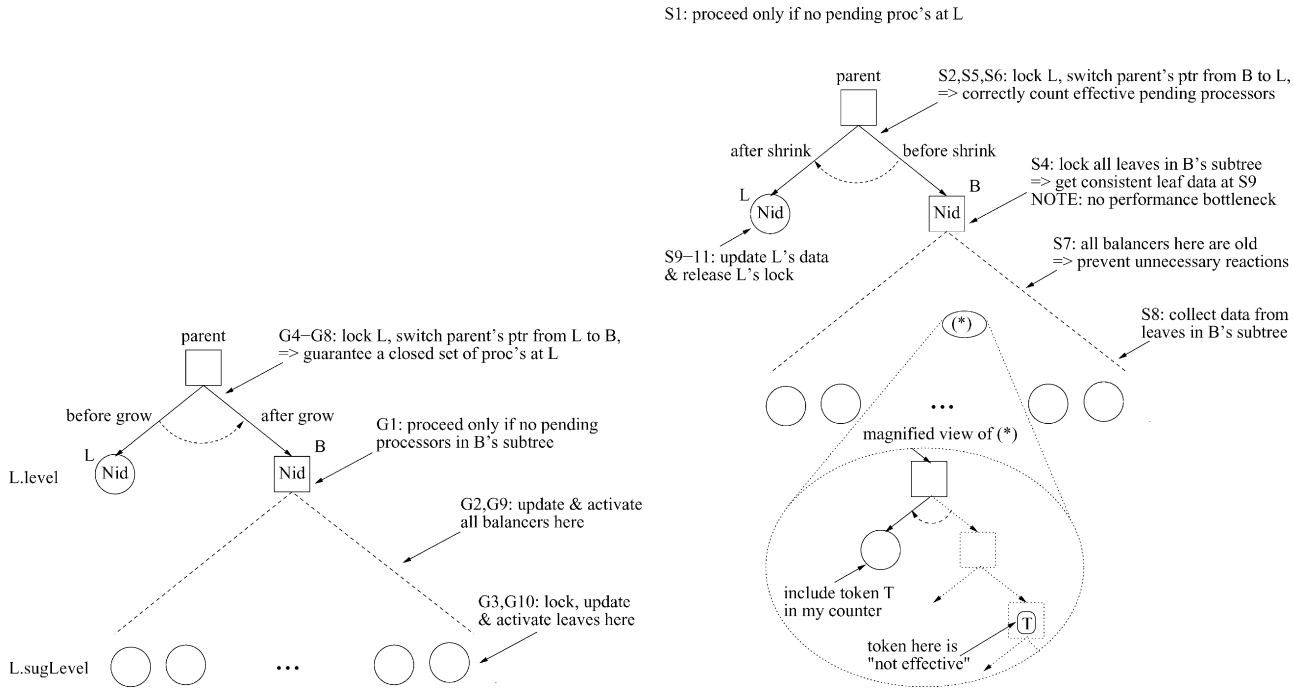

Fig. 8. Illustrations for *Grow* and *Shrink* procedures.

NEXTCOUNTG($ppL, CurCount$)

$next\_count\_value := CurCount + ppL * 2^{L.level}$; $increment = next\_count\_value - RightmostLeaf.init$;

for *every leaf L' at level L.newLevel* **do** $L'.count = L'.init + increment$;

Fig. 9. The *NextCountG* function in the *Grow* procedure.

tokens in T (step G1). Otherwise, it will cause *old* tokens to get *new* values, which will consequently cause *omissions* in the sequence of numbers received by tokens at the end. When the condition for expansion is satisfied, the expansion will be activated by subsequent tokens visiting L since L is under high contention. Consistency is ensured by locking L (step G4) and by switching the pointer from L to B (step G5), which leaves a *closed* set of processors in L. The lock acquisition is *conditional*, i.e. if one of L's ancestors is holding a lock, L's attempt to acquire the lock will return *Fail*. In such a situation the expansion procedure aborts, since a failure to acquire the lock means that there is an overlapping shrinkage operation being executed by one of L's ancestors. Finally, the state of all balancers in the new subtree is set to *ACTIVE* for reactive adjustments (steps G9, G10). Note that overlapping growth operations by one of L's ancestors must abort due to the existence of the token/processor at L that is carrying out the growth procedure (step G1) (Figs. 7 and 8).

In order to minimize the adjustment delay for processors, we need to minimize locking intervals for *working* leaves (i.e. leaves which execute the grow procedure). Toward this, the *Grow* procedure acquires necessary leaves in *decreasing* order of their labels so as to acquire the *working* leaf L last, and the *Shrink* procedure acquires necessary leaves in *increasing* order of their labels so as to acquire the *non-working* leaf L last. Moreover, since the tree allows concurrent adjustments at any level, adjustments at high levels should have higher priorities than those at lower levels when there is collision between the adjustments. Deadlocks due to potential interferences between growth and shrinkage procedure are avoided by the *conditional lock acquisition* (cf. also the proof in Section 6): two such potentially interfering procedures will have to compete for the same conditional-lock; when this happens, one of them will win and proceed and the other will lose and back-off (thus releasing also the previously acquired locks).

5.2.1. Computing proper values for new leaves

Since toggle-bits of balancers in the new subtree are reset to 0, the first token traversing the new subtree will arrive at the rightmost leaf at level *L.newLevel*. Therefore, counting values for the new leaves (step G10) are computed as in Fig. 9.

L's current counting value *CurCount* and the number of pending processors *ppL* are read at steps G7 and G6, respectively. The *next_count_value* variable is the next counting value after all the pending processors leave the leaf L. The field *init* of *LeafType* (Fig. 5) is the base to calculate counting values and is unchanged. For instance, in Fig. 1 the base *init* of leaf $C6$ is 1 and the nth counting value is $1 + n * 4$.

```
int NextCountS ( int eppB, list_t SL, list_t SLCount)
    Convert leaves in T to the same level, the lowest level ⇒ new sets of leaves SL' & counting values SLCount';
    Distribute the number of effective processors eppB on the leaves in SL' so that step-property is satisfied.;
    Call the leaf last visited by the pending processors lastL;
    return (lastL.count − 2^(lastL.level) + 2^(B.level));
```

Fig. 10. The *NextCountS* function in *Shrink* procedure.

5.3. Implementing the shrink procedure

A processor p visiting a leaf L_0, whose suggested reaction is to shrink to level $L_0.newLevel$, adds L_0's vote[4] to the vote boxes of all balancers on p's path from level $L_0.newLevel$ down to level $L_0.level - 1$. If L_0's new suggested level $L_0.newLevel$ is lower than its old suggested level $L_0.oldLevel$, which is updated to $L_0.level$ after each shrinking adjustment, L_0 removes its old votes at levels above its new suggested level (step E1 in Fig. 7). If L_0's new suggested level is higher than its old suggested level, it adds its votes at levels from $L_0.newLevel$ to $L_0.oldLevel$ (steps E2, E3). In the vote-adding procedure, when reaching a balancer B with enough votes, the processor starts a shrink procedure at the balancer (step E5). The procedure is illustrated in Fig. 8 and its pseudocode is in Fig. 7.

Symmetrically to the growth procedure, the procedure of shrinking a subtree T rooted at balancer B to B's matching leaf L needs to properly set L's counting value so as to ensure the counting property. The shrinkage procedure occurs only if there are no pending tokens in L. Otherwise, it will cause *old* tokens to get *new* values (step S1 in *Shrink*). The procedure locks L (step S2) and properly sets L's counting value (step S9). The value is computed from a consistent measurement of the number of pending processors in T and the counting values of leaves L' in T (cf. Fig. 10). Consistency is ensured by locking leaves L' in T (step S4) and switching the pointer from B to L, which leaves a *closed* set of processors in T. Similarly to the *Grow* procedure, the lock acquisition is conditional. The leaves in T are locked in an *increasing* order of their labels, to keep the locking periods of working leaves minimal and to avoid deadlocks, as explained in the context of the *Grow* procedure. Note that an overlapping shrinkage procedure by one of L's ancestors cannot cause any of the attempts to lock L' at step $S4$ to fail, since the overlapping procedure must first lock L successfully (if the overlapping procedure had succeeded, it would have caused the shrinkage from B to L to abort earlier, at step S2). We will prove that the growth and shrinkage procedure do not interfere with each other in Section 6. Finally, the shrinkage procedure sets state of balancers and leaves in T to *OLD* (steps S7, S8).

[4] Recall that since the subtree rooted at L_0's ancestor at level $L_0.newLevel$ contains other leaves that may have not recommended to shrink to l_{higher}, our algorithm uses asynchronous voting to coordinate them.

5.3.1. Computing a proper value for the new leaf

The counting value for the new leaf is computed using the set of active leaves *SL*, their counting values *SLCount* and the number of effective pending processors *eppB* in the subtree T (step S9). The value is the result of the *NextCountS* function in Fig. 10.

For instance, in Fig. 3 if the subtree of balancer 2 shrinks to leaf 2 and the set of active leaves *SL* is {4, 10, 11}, leaf 4 needs to be converted to two leaves 8 and 9 at the same level as leaves 10 and 11, the lowest level. Thus, the new set of leaves SL' is {8, 9, 10, 11}. After converting, the subtree becomes balanced and the step-property must be satisfied on the subtree. The following property of trees satisfying the step-property was exploited to distribute the set of effective processors *eppB* to leaves in SL': if *MaxValue* denotes the highest counting value at the lowest level, the counting values of leaves must be in range $(MaxValue - 2^{\text{LowestLevel}}, MaxValue]$.

5.4. Efficiency considerations in synchronization

To allow for even higher parallelism in traversing the ST-trees, we define and implement two advanced synchronization operations: *read-and-follow-link* and *conditional lock acquisition*. The former is a lock-free operation caring for maximizing parallelism at balancers. The latter cares for minimizing disturbance due to reactive adjustments. The operations are described in this subsection and their pseudocode is in Fig. 11.

The read-and-follow-link operation: In order to support high parallelism, we design a lock-free synchronization procedure, instead of mutual exclusion, at balancers, where the collision between visiting processors is high. The method allows all processors to concurrently traverse a balancer, reducing their traversal latency. Note that in designing this, we need to ensure that the number of pending processors can be counted correctly by the adjustment procedures, in order to assign correct initial values to the leaves/counters of the (sub)tree affected by the adjustment, for the ST-tree to satisfy the counting property.

Fig. 12 illustrates the potential problem of incorrectly counting the number of pending processors. A processor p_2 reads a pointer *ptr* that shows which node the processor is going to visit. Before the processor updates its new location in *Tracing*[2], another processor p_1 executes a reactive adjustment that switches the pointer to another position and counts the number of pending processors in the old branch via

```
type NodeType = record union
                    val: record Nid : [1..MaxNodeId]; kind : {BALANCER, LEAF}; end;
                    ptr : *NodeType; end;
                mask: bit; end; /*in one word*/

BasicAssign(NodeType * trace_i, NodeType * child)
A0  *trace_i :=< child, 0 >;/*clear mask-bit*/
A1  temp := *child; /*get the expected value*/
A2  temp.mask := 1; /*set the mask-bit*/
A3  CAS(trace_i, < child, 0 >, temp);

NodeType Assign(NodeType * trace_i, NodeType * child)
AR0 BasicAssign(trace_i, child);
AR1 return Read(trace_i);

boolean AcquireLock_cond( int lock, int Nid)
AL0 while ((CurOccId := CAS(lock, 0, Nid)) ≠ 0) do
AL1     if IsParent(CurOccId, Nid) then return Fail;
AL2     Delay using exponential backoff;
AL3 return Success;

NodeType Read(NodeType * trace_i)
R0  do
R1      local := *trace_i;
R2      if local.mask = 0 then /*trace_i is marked*/
R3          temp := *local.ptr; /*help Assign() ...*/
R4          temp.mask := 1;
R5          CAS(trace_i, local, temp);
R6  while(local.mask = 0); /*... until it completes*/
R7  return local;

Release( int lock)
    lock := 0;
```

Fig. 11. The BasicAssign, Assign, Read, and AcquireLock_cond operations.

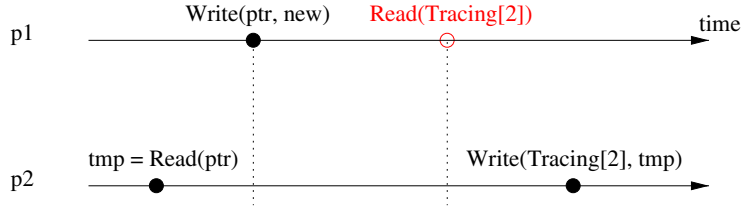

Fig. 12. An illustration for the need of *read-and-follow-link* operation.

Tracing array. Since p_1 reads an obsolete value of *Tracing*[2], it does not count p_2. Then, p_2 follows the old value of *ptr* and visits the old branch. That means the number of pending processors in the old branch is higher than what p_1 has counted, causing the tree to compute incorrect values for the new leaves in the *Grow* and *Shrink* procedures (steps G10 and S9 in Fig. 7).

Intuitively, observe that if both *Read*(*ptr*) and *Write*(*Tracing*[2]) on p_2 occur atomically to *Read*(*Tracing*[2]) on p_1, the problem is solved. This motivated us to design the operations called *BasicAssign* and *Read* (cf. Fig. 11). Each element *Tracing*[i] can be updated by only one processor p_i via the *BasicAssign* operation and can be read by many other processors via the *Read* operation. The former reads the variable pointed by *child* and then writes its value to the variable pointed by $trace_i$. The latter reads the variable at $trace_i$. The *BasicAssign*() and *Read*() operations are lock free [13] and linearizable [16]. [5]

Conditional lock-acquisition operation: During both the growth and shrinkage procedures, a processor acting on behalf of a node *Nid* must invoke *AcquireLock_cond* to acquire a leaf lock *lock* by writing the node label *Nid* to the lock. If the lock is occupied by one of *Nid*'s ancestors, the procedure returns *Fail* (line AL1, Fig. 11). The procedure is built using the busy-waiting with exponential backoff technique instead of queue-lock because the contention on the leaves is kept low. In a low-contention environment, the busy-waiting with exponential back-off technique achieves better throughput than the queue-lock, which requires more complicated data structures [21].

[5] The proof is given in Lemma 5.

For acquiring leaf locks to increase counting values (i.e. not to perform adjustment tasks), processors invoke the *AcquireLock* procedure without condition. The procedure is similar to *AcquireLock_cond* but does not check the ancestral condition (line AL1, Fig. 11). The *AcquireLock* procedure always returns *Succeed*.

The way these locking mechanisms ensure safety and liveness for the tree accesses is explained in the descriptions of the *Grow* and *Shrink* procedures and is proven in Section 6.

6. Correctness proof

We first prove that the *BasicAssign* and *Read* operations (cf. Fig. 11) are lock free and linearizable [16,13] as mentioned in Section 5.4.

Lemma 5. *The* BasicAssign *and* Read *operations are linearizable and lock-free*.

Proof. In the *BasicAssign* operation, the variable pointed by $trace_i$ is first locked by clearing the *mask* bit. If the *mask* bit of the $*trace_i$ variable is zero, the variable is locked and its value is the address of another variable whose value must be written to $*trace_i$. After reading the expected value $*child$, the operation sets the *mask* bit to 1 so as to unlock $*trace_i$ and subsequently writes the value to $*trace_i$ using a *compare_and_swap* primitive (lines A1–A3). If $*trace_i$ is still containing the pointer *child*, the primitive will successfully write the value to $*trace_i$. Otherwise, another operation has helped the operation complete the assignment.

When reading the value of a variable $*trace_i$, the *Read* operation checks if the variable is locked (line R2). If the *mask* bit of the variable is zero, the operation will help the corresponding *BasicAssign* operation to write the expected value to the variable before trying to read it again (lines R3–R5). The linearization point [16] of the *Read* operation is the point it reads a value with a non-zero *mask* bit (line R1); the linearization point of the *BasicAssign*() operation is the point it writes a pointer with a zero *mask* bit to $*trace_i$ (line A0).

The *BasicAssign* and *Read* operations are lock free since the *Read* operation retries only if there has been an operation with a successful *CAS* at line A3 or R5, which is progressing. □

In the implementation of the self-tuning tree on real computer systems, we can use the last bit of a pointer, which is unused because of word-alignment memory architecture, as the *mask* bit.

Second, since leaves are locked in decreasing order of leaf identities in the *Grow* procedure but in increasing order in the *Shrink* procedure, we need to prove that deadlock never occurs due to interferences between the *Grow* and *Shrink* procedures.

Lemma 6. *The self-tuning tree is deadlock-free*.

Proof. Interferences between two balancers that are trying to lock leaves[6] occur only if one of the balancers is the other's ancestor in the tree. Assume that there are two balancers b_i and b_j, where b_i is b_j's ancestor.

- *Case 1:* If both balancers b_i and b_j execute the *Shrink* procedures that shrink their subtrees to leaves, both will lock leaves in increasing order of leaf identities by using the *AcquireLock_cond* operation. If the leaf with smallest identity that b_j needs is locked by b_i, the operation called by b_j will return *Fail* immediately. This is because the leaf is locked by an ancestor of b_j. If the leaf is locked by b_j, b_i must wait at the leaf until b_j completes its own work and then b_i continues locking necessary leaves. If there are no crashes of processors locking the leaves on behalf of a balancer, no deadlock will occur.
- *Case 2:* b_i executes the *Shrink* procedure, which shrinks its subtree to a leaf, and b_j executes the *Grow* procedure, which expands its matching leaf to a subtree. In this case, b_i tries to lock all necessary leaves in increasing order of leaf identities and b_j does that in decreasing order of leaf identities. Assume that b_i has locked leaf k successfully and is now trying to lock leaf $(k+1)$ whereas b_j has locked leaf $(k+1)$ successfully and is trying to lock leaf k. Because b_i and b_j use the *AcquireLock_cond* procedure that conditionally acquires the locks, b_j will fail to lock leaf k, which is being locked by its ancestor, and will release all the leaves it has locked so far (lines G3 and G4, Fig. 7). Eventually, b_i successfully locks leaf $k+1$ and continues locking other necessary leaves. That is, deadlock does not occur in this case either.

Note that b_i cannot execute the *Grow* procedure due to the step $G1$: there is a pending processor in b_i's subtree that helps b_j execute its adjustment. □

Corollary 7. *In the Shrink procedure, if the corresponding balancer has successfully locked the necessary leaf with smallest identity, it will successfully lock all the leaves it needs*.

Therefore, the *Shrink* procedure returns *Fail* only if the matching leaf of the corresponding (active) balancer is being locked by an ancestor of the balancer.

Lemma 8. *There is no interference between any two Grow procedures in the self-tuning tree*.

Proof. Similarly to the proof of the previous lemma: (i) the interference between two balancers who are trying to lock leaves occurs only if one of the balancers is the other's ancestor in the tree and (ii) there is no case that two expansion phases are executed at the same time and one of the two corresponding balancers is an ancestor of the other. □

Lemma 8 explains why the *Grow* procedure does not lock balancers before resetting their variables (line G2, Fig. 7).

Finally, we prove that the number of processors used to calculate counting values for new leaves in both *Grow* and *Shrink* procedures is calculated accurately via the global array *Tracing*.

[6] Recall that processors lock leaves on behalf of balancers.

Definition 9. *Old balancers/leaves* are the balancers/leaves whose state is OLD.

Definition 10. *Effective processors/tokens* are the processors/tokens that are visiting neither old balancers nor old leaves of a locked subtree.

Only the *effective processors* affect the next counting values calculated for new leaves. An illustration to enhance the understanding of this definition is given in Fig. 8-*Shrink*, where the token marked as "T" at the lower part of the figure is not effective.

Lemma 11. *The number of* effective processors *that are pending in a locked subtree or visiting a locked leaf is calculated accurately in the Grow and Shrink procedures.*

Proof. A processor p_i executing an adjustment task switches a pointer from one branch of the tree to the other before counting pending processors in the old branch (lines G5, G6 in *Grow* and lines S5, S6 in *Shrink*, Fig. 7). Since the *BasicAssign* and *Read* operations are linearizable (by Lemma 5), p_i counts the number of pending processors accurately. Recall that the old branch is locked as a whole so that no processor can leave the tree from the old branch as well as no other adjustment can concurrently take place in the old branch until the counting completes. The set of pending processors in the old branch consists of both *effective* and *ineffective* processors.

In the case of tree expansion, the number of pending processors at the locked leaf is the number of effective processors.

In the case of tree shrinkage, we lock both old and active leaves of the locked subtree so that no pending processor in the subtree can switch any pointers. Recall that to switch a pointer, a processor has to successfully lock the leaf corresponding to that pointer. On the other hand, (i) we set states of all balancers and leaves in a locked subtree/locked leaf to *Old* before releasing them (lines S7, S8 in *Shrink* and line G7 in *Grow*), and (ii) after locking necessary balancers and leaves, processors continue executing the corresponding *Shrink*/*Grow* procedures only if the switching balancers/leaves are still in an active state (line G4 in *Grow* and line S2 in *Shrink*). Corollary 7 implies, for the *Shrink* procedure, that checking the switching-balancer state after locking the necessary leaf with smallest identity (line S2) guarantees that no ancestor can change the switching-balancer state from *Active* to *Old* afterwards. Therefore, a pending processor in the locked subtree that has visited an *Old* balancer or an *Old* leaf will never visit an *Active* one in this locked subtree. Similarly, a pending processor in the locked subtree that has visited an *Active* balancer or an *Active* leaf will never visit an *Old* one in this locked subtree. Hence, by checking the state of the node that a pending processor p_j in the locked subtree is currently visiting, we can know whether the processor p_j is effective (line S6 in *Shrink*). In conclusion, the number of effective processors in a locked subtree is calculated accurately in the *Shrink* procedure in the case of tree shrinkage. □

Since the number of effective pending processors is counted accurately (by Lemma 11), the counting values that are set for leaves after adjustment steps are correct.

Theorem 12. *The ST-tree correctly implements a counting tree. It also provides an online algorithm to reactively decide on new sizes based on the observed contention, guaranteeing that: in each contention-rising/contention-dropping phase, the reactive decision is competitive with competitive ratio* $c = \varphi - \frac{\varphi-1}{\varphi^{1/(\varphi-1)}} = \Theta(\ln \varphi)$, *where* $\varphi = \frac{P}{2}$ *and P is the maximum number of processors that can access the tree simultaneously.*

7. Experimental study

In this section, we study the behavior of the new self-tuning tree. We implemented two versions of the tree: the *ST-tree(P)*, that uses the elimination technique [24,23] to alleviate contention on toggle-bits as the original reactive diffracting tree does, and the *ST-tree*, that uses a low-contention hardware primitive *fetchop* supported by the SGI Origin2000 [18] instead.

We used the original reactive diffracting tree [8] as a basis of comparison since it is one of the most efficient reactive counting constructions in the literature. The most difficult issue in implementing the tree is to find the best folding and unfolding thresholds as well as the number of consecutive timings called *UNFOLDING_LIMIT*, *FOLDING_LIMIT* and *MINIMUM_HITS* in [8]. Subsection *Load Surge Benchmark* in [8] described that the tree had a depth 3 when it was run for the index-distribution benchmark [24] with 32 processors, the highest possible load ($work = 0$) and the number of consecutive timings set to 10. Following the description, we tuned the thresholds for the tree on the ccNUMA Origin2000 system with 30 processors. The result is that the folding and unfolding thresholds are 3 and 10 μs, respectively. This selection of parameters did not only keep our experiments consistent with the ones presented in [8] but also gave the best performance for the reactive diffracting tree on our system. Regarding the prism size, the algorithmic construction to implement the elimination technique, each node has $c2^{(d-l)}$ prism locations, where $c = 0.5$, d is the average value of the tree depths estimated by processors traversing the tree and l is the level of the node [23,8]. The upper bound *MAXSPIN* for the adaptive spin is 128 as mentioned in [24].

We used a full-contention benchmark and a surge-load benchmark that are similar to the index-distribution benchmark with $work = 0$ and the surge-load benchmark in [8]. The benchmarks ran on a ccNUMA SGI Origin 2000 with 30 250 MHz MIPS R10000 processors running IRIX 6.5. In order to make these empirical results accessible to other researchers and practitioners, the C code for the tested algorithms is available at http://www.cs.chalmers.se/~phuong/satNov05.tar.gz.

Our primary interest has been to verify whether the self-tuned ST-tree (with or without the elimination) can make similar reactive decisions to the hand-optimized RD-tree. The study confirms that, as well as it shows that the above is achieved

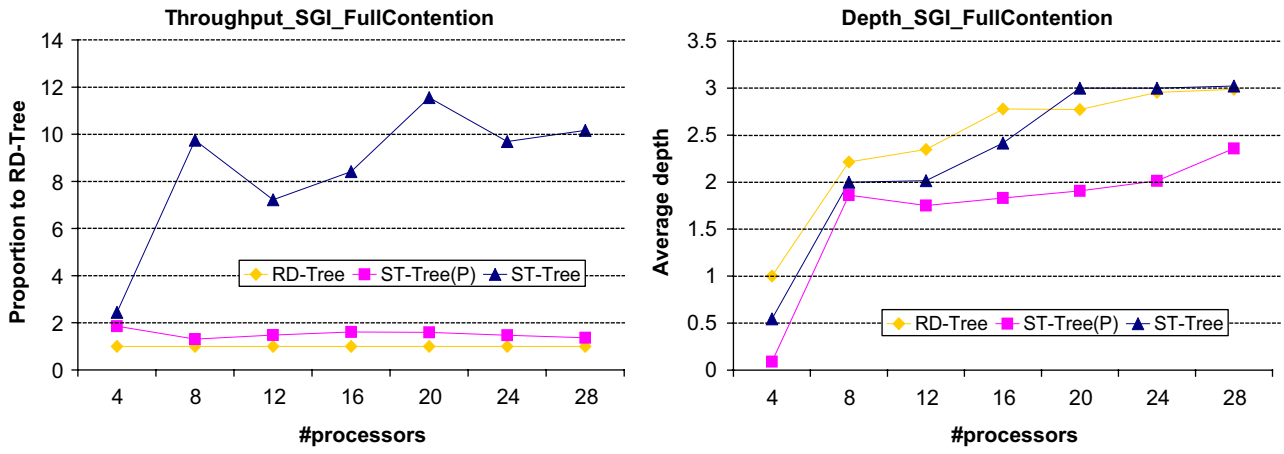

Fig. 13. Throughput and average depth of trees in the full-contention benchmark on SGI Origin2000.

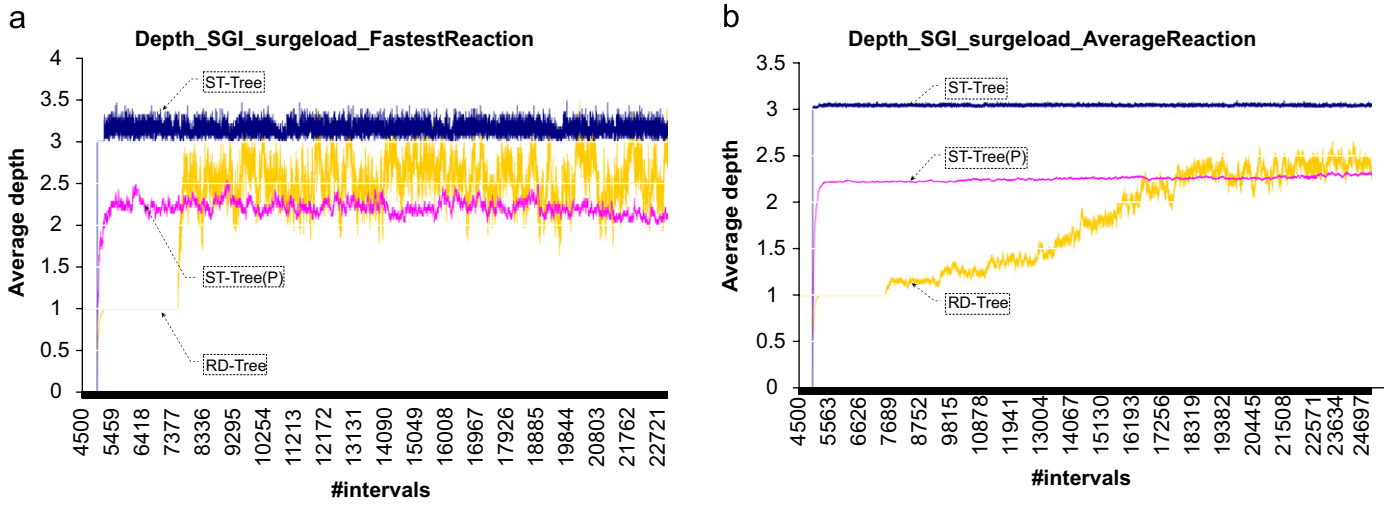

Fig. 14. Average depths of trees in the surge load benchmark on SGI Origin2000, the fastest and the average reactions.

with at least as good performance as the RD-tree. Moreover, the study highlights other benefits of the ST-tree algorithm (with or without the elimination). These are explained in the subsequent sections.

7.1. Full contention benchmark

In the benchmark, each processor continuously traverses the respective tree and gets a counting value. The number of processors participating in the benchmark varied from 4 to 28, which simulates different contention levels on the tree. We ran the benchmark for 1 min and measured the average size of the tree as well as the average number of the *TraverseTree* operations (or the number of tokens passing the tree) in 1 s. The results are shown in Fig. 13, where the right chart shows the tree's average depth and the left chart shows the proportion of the ST-tree's throughput to that of the RD-tree. The tree with higher throughput is better.

In the left chart we observe that ST-tree(P) and ST-tree perform better. In the case of 28 processors, ST-tree(P), which uses the same elimination technique as RD-tree, is 36% faster than RD-tree and ST-tree is 10 times faster. Since each tree *continuously* adjusts its size around the average value due to contention variation on its leaves even in the case that the number of participating processors is fixed (cf. Fig. 14), the tree with more efficient adjustment will achieve better performance in the full-contention benchmark. The reactive adjustments of the ST-tree(P)/ST-tree and RD-tree have algorithmic differences:

- The former reacts to contention variation faster due to its low-overhead multi-level adjustment as described in Sections 3 and 5.4. In the latter, leaves shrink or grow only one level in one adjustment step and then have to wait for a given number of incoming processors before continuing shrinking or growing.

- In the latter, whenever a leaf shrinks or grows, all processors visiting the leaf are blocked completely until the reaction process completes. Moreover, some processors may be forced to go back to higher nodes (possibly several times) before exiting the tree. In the former, this problem is avoided by the introduction of matching leaves, which provides high parallelism.

Another observation is that when the contention increases, the self-tuning ST-tree automatically adjusts its size close to that of the hand-tuned RD-tree (cf. the right chart). The result implies that the longer lock-based adjustment at the leaves of the RD-tree blocks more processors at the leaves, causing contention on them as high as that on the ST-tree's leaves, whose throughput is 10 times higher. The size difference between ST-tree and ST-tree(P) implies that the elimination technique [24,23] which is a useful construction to alleviate contention on toggle-bits if other means are not available, may also delay processors at balancers, consequently reducing contention on leaves, but at the cost of lower throughput.

7.2. Surge load benchmark

The benchmark shows how fast the trees react to contention variation. In this benchmark, we measured the average depth of each tree in each interval of 400 µs. The measurement was done by a monitor processor. At interval 5000, the number of processors was changed from 4 to 28. The depth of each tree at the interval 5001 was measured after all the new processors were synchronized with the monitor processor, i.e. the period between the end of interval 5000 and the beginning of interval 5001 was not 400 µs.

Fig. 14 shows the depths from interval 4500 to 25 000. The left chart shows the fastest-reaction experiment for each tree over 15 experiments. It also shows the amplitude within which the tree size varied when the number of processors is fixed to 28. The RD-tree's amplitude is shown to be the largest. The right chart shows the average reaction time for each tree over 15 experiments. In the case of 28 processors, the ST-tree reached depth 3 at interval 5009, i.e only after 9 intervals since the time all 28 processors started to run. The ST-tree(P) reached a depth 2.2 at interval 5362 and the RD-tree reached level 2.4 at interval 17 770. The difference between the average reaction times of ST-tree and ST-tree(P) implies that the elimination technique delays processors at balancers when the load surges, making contention at leaves increase only gradually (at the cost of potentially lower throughput, though, as discussed earlier in this section).

The difference between ST-tree(P) and RD-tree reaction delays confirms the advantage of the fast multi-level adjustment scheme used in ST-tree/ST-tree(P). It is interesting to note that whatever load patterns are used, the *adjustment interval* between the time a tree makes an adjustment decision and the time the tree completes the adjustment should better be as short as possible. The reason for that was outlined in Section 3.1 and it is also illustrated in the following experiment, which studies the throughput of the trees when the load changes suddenly and frequently. We extended the benchmark so that the number of processors accessing the trees repeatedly changed from 4 to 28 or from 28 to 4 for each period of 0.1 s (i.e. every 25 intervals—recall that on average it takes the ST-tree some 9 intervals to adjust, the ST-tree(P) approximately 500 intervals to adjust and the RD-tree approximately 12 000 intervals to adjust). The benchmark was run in 1 min or in 600 cycles. We measured the average number of the *TraverseTree* operations, i.e. the number of tokens passing a tree, in 1 s. The result is shown in Fig. 15. As expected, the ratio of ST-tree/ST-tree(P)'s throughput to RD-tree's throughput in the benchmark is higher than in the full-load benchmark in which contention on leaves varies slightly due to the number of processors being fixed. In the surge-load benchmark, the ST-tree(P)'s throughput is 2.4 times higher than that of RD-tree and ST-tree's throughput is 15.3 times higher.

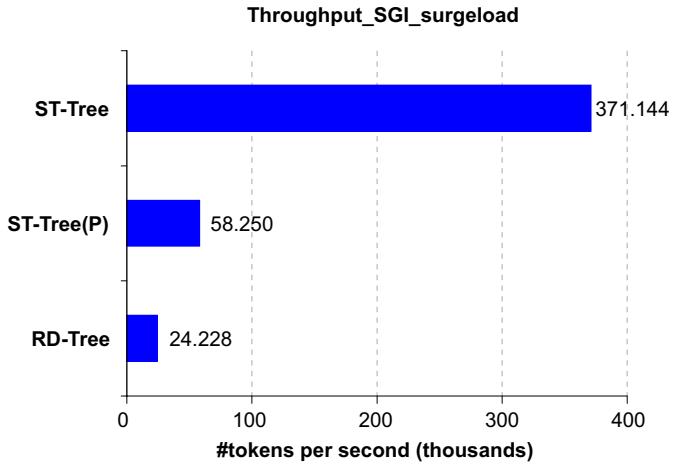

Fig. 15. Throughput of trees in the surge load benchmark on SGI Origin2000.

8. Conclusion

This paper has presented the new *self-tuning* reactive diffracting tree, a data object that distributes concurrent memory accesses to different banks in a coordinated manner. The tree extends the original *hand-tuned* reactive diffracting tree, a successful result in the area of reactive concurrent data structures, in several aspects. To circumvent the need of hand-tuned parameters, the trade-off between the tree depth and the contention on leaves has been analyzed as an online problem and subsequently an efficient online solution has been suggested. The solution, which was inspired by an optimal online algorithm for an online financial problem [10], helps the self-tuning tree make precise reactive decisions. A considerable factor for the ST-tree efficiency is the new data-structure that allows the self-tuning tree to freely grow and shrink by several levels in one adjustment step. The construction provides high parallelism, reducing the overhead of reactive adjustments. The construction has space complexity comparable with that of the original hand-tuned tree. It exploits low-contention occasions on subtrees to make its

locking process as efficient as in the original hand-tuned tree although the locking process locks more nodes at the same time. As a result, the new self-tuning tree reacts quickly to contention variation and offers better latency for the traversing tokens. The experimental studies on the SGI Origin2000, a commercial ccNUMA multiprocessor, confirm these results in the relative study of the *self-tuning* tree and the original *hand-tuned* tree.

The paper provides a starting point to design reactive distributed data structures using competitive analysis. The challenge of designing such a *distributed* data structure is that each of its components must be able to adaptively make appropriate decisions using only *local* information in *unpredictable* environments. To deal with the environmental uncertainty, competitive analysis seems to be a promising approach as demonstrated in this paper. An analysis that considers other aspects, such as reaction delays, monitoring periods and appropriate adversary models is open for further research.

References

[1] A. Agarwal, M. Cherian, Adaptive backoff synchronization techniques, in: Proceedings of the Annual International Symposium on Computer Architecture, 1989, pp. 396–406.

[2] M. Ajtai, J. Aspnes, C. Dwork, O. Waarts, A theory of competitive analysis for distributed algorithms, in: IEEE Symposium on Foundations of Computer Science, 1994, pp. 401–411.

[3] T.E. Anderson, The performance of spin lock alternatives for shared-money multiprocessors, IEEE Trans. Parallel Distrib. Syst. 1 (1) (1990) 6–16.

[4] J. Aspnes, M. Herlihy, N. Shavit, Counting networks and multi-processor coordination, in: Proceedings of ACM Symposium on Theory of Computing (STOC), 1991, pp. 348–358.

[5] J. Aspnes, M. Herlihy, N. Shavit, Counting networks, J. ACM 41 (5) (1994) 1020–1048.

[6] G. Barnes, A method for implementing lock-free shared-data structures, in: Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA), 1993, pp. 261–270.

[7] E.A. Brewer, C.N. Dellarocas, A. Colbrook, W.E. Weihl, Proteus: a high-performance parallel architecture simulator, Technical Report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1991.

[8] G. Della-Libera, N. Shavit, Reactive diffracting trees, J. Parallel Distrib. Comput. 60 (7) (2000) 853–890.

[9] E.W. Dijkstra, The mathematics behind the Banker's algorithm, in: Selected Writings on Computing: A Personal Perspective, Springer, Berlin, 1982, pp. 308–312.

[10] R. El-Yaniv, A. Fiat, R.M. Karp, G. Turpin, Optimal search and one-way trading online algorithms, Algorithmica 30 (1) (2001) 101–139.

[11] J.R. Goodman, M.K. Vernon, P.J. Woest, Efficient synchronization primitives for large-scale cache-coherent multiprocessors, in: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1989, pp. 64–75.

[12] G. Graunke, S. Thakkar, Synchronization algorithms for shared-memory multiprocessors, IEEE Comput. 23 (6) (1990) 60–69.

[13] M. Herlihy, Wait-free synchronization, ACM Trans. Programming Syst. 11 (1) (1991) 124–149.

[14] M. Herlihy, A methodology for implementing highly concurrent data objects, ACM Trans. Programming Languages Syst. 15 (5) (1993) 745–770.

[15] M. Herlihy, S. Tirthapura, R. Wattenhofer, Competitive concurrent distributed queuing, in: Twentieth ACM Symposium on Principles of Distributed Computing (PODC), Newport, Rhode Island, 2001.

[16] M.P. Herlihy, J.M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Trans. Programming Languages Syst. 12 (3) (1990) 463–492.

[17] A.R. Karlin, K. Li, M.S. Manasse, S. Owicki, Empirical studies of competitive spinning for a shared-memory multiprocessor, in: Proceedings of the ACM Symposium on Operating Systems Principles, 1991, pp. 41–55.

[18] J. Laudon, D. Lenoski, The sgi origin: a ccnuma highly scalable server, in: Proceedings of the Annual International Symposium on Computer Architecture (ISCA-97), 1997, pp. 241–251.

[19] N. Lynch, N. Shavit, A. Shvartsman, D. Touitou, Timing conditions for linearizability in uniform counting networks, Theoret. Comput. Sci. 220 (1) (1999) 67–91.

[20] M. Mavronicolas, M. Papatriantafilou, P. Tsigas, The impact of timing on linearizability in counting networks, in: Proceedings of the International Symposium on Parallel Processing (IPPS), 1997, pp. 684–688.

[21] J.M. Mellor-Crummey, M.L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, ACM Trans. Comput. Syst. 9 (1) (1991) 21–65.

[22] N. Shavit, D. Touitou, Elimination trees and the construction of pools and stacks: preliminary version, in: Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA), 1995, pp. 54–63.

[23] N. Shavit, E. Upfal, A. Zemach, A steady state analysis of diffracting trees, Theory Comput. Systems 31 (4) (1998) 403–423.

[24] N. Shavit, A. Zemach, Diffracting trees, ACM Trans. Comput. Syst. 14 (4) (1996) 385–428.

[25] N. Shavit, A. Zemach, Combining funnels: a new twist on an old tale, in: Proceedings of ACM Symposium on Principles of Distributed Computing (PODC), 1998, pp. 61–70.

[26] D.D. Sleator, R.E. Tarjan, Amortized efficiency of list update and paging rules, Commun. ACM 28 (2) (1985) 202–208.

[27] R. Wattenhofer, P. Widmayer, An adaptive distributed counting scheme, in: Proceedings of International Colloquium on Structural Information and Communication Complexity (SIROCCO), 1998, pp. 145–157.

[28] R. Wattenhofer, P. Widmayer, The counting pyramid: an adaptive distributed counting scheme, J. Parallel Distrib. Comput. 64 (4) (2004) 449–460.

[29] P.-C. Yew, N.-F. Tzeng, D.H. Lawrie, Distributing hot-spot addressing in large-scale multiprocessors, IEEE Trans. Comput. 36 (4) (1987) 388–395.

Phuong Hoai Ha received the BEng degree from the Department of Information Technology, Ho-Chi-Minh City University of Technology, Vietnam and the Ph.D. degree from the Department of Computer Science and Engineering, Chalmers University of Technology, Sweden. His research interests are online algorithms and parallel/distributed computing, including reactive synchronization, fault-tolerant coordination and concurrent data structures (www.cs.chalmers.se/∼phuong).

Marina Papatriantafilou is an associate professor at the Department of Computing Science, Chalmers University of Technology, Sweden. She received the BSc and Ph.D. degrees from the Department of Computer Engineering and Informatics, University of Patras, Greece. She has also worked at the National Research Institute for Mathematics and Computer Science in the Netherlands (CWI), Amsterdam and at the Max-Planck Institute for Computer Science, (MPII) Saarbruecken, Germany. She is interested in research on distributed and multiprocessor computing, including synchronization, communication/coordination, networking, scalability, real-time and fault-tolerance aspects.

Philippas Tsigas' research interests include concurrent data structures for multiprocessor systems, communication and coordination in parallel systems, fault-tolerant computing, mobile computing. He received a BSc in Mathematics from the University of Patras, Greece and a Ph.D. in Computer Engineering and Informatics from the same University. Philippas was at the National Research Institute for Mathematics and Computer Science, Amsterdam, the Netherlands (CWI), and at the Max-Planck Institute for Computer Science, Saarbrücken, Germany, before. At present he is an associate professor at the Department of Computing Science at Chalmers University of Technology, Sweden (www.cs.chalmers.se/∼tsigas).