Self-tuning Reactive Distributed Trees for Counting and Balancing

Phuong Hoai Ha, Marina Papatriantafilou, and Philippas Tsigas

Department of Comp. Science, Chalmers University of Technology, SE-412 96
Göteborg, Sweden. [phuong|ptrianta|tsigas]@cs.chalmers.se

Abstract. The main contribution of this paper is that it shows that it *is* possible to have reactive distributed trees for counting and balancing with no need for the user to fix manually any parameters. We present a data structure that in an on-line manner balances the trade-off between the tree traversal latency and the latency due to contention at the tree nodes. Moreover, the fact that our method can expand or shrink a subtree several levels in any adjustment step, has a positive effect in the efficiency: this feature helps the self-tuning reactive tree minimize the adjustment time, which affects not only the execution time of the process adjusting the size of the tree but also the latency of all other processes traversing the tree at the same time with no extra memory requirements. Our experimental study compared the new trees with the reactive diffracting ones on the SGI Origin2000, a well-known commercial ccNUMA multiprocessor. This study showed that the self-tuning reactive trees i) select the same tree depth as the reactive diffracting trees do; ii) perform better and iii) react faster.

1 Introduction

Distributed data structures suitable for synchronization that perform efficiently across a wide range of contention conditions are hard to design. Typically, "small", "centralized" such data structures fit better low contention levels, while "bigger", "distributed" such data structures can help in distributing concurrent processor accesses to memory banks and in alleviating memory contention.

Diffracting trees [1] are distributed data structures. Their most significant advantage is the ability to distribute a set of concurrent process accesses to many small groups locally accessing shared data, in a coordinated manner. Each process(or) accessing the tree can be considered as leading a *token* that follows a path from the root to the leaves. Each node is a computing element receiving tokens from its single input (coming from its parent node) and sending out tokens to its outputs; it is called *balancer* and acts as a *toggle mechanism* which, given a stream of input tokens, alternately forwards them to its outputs, from left to right (sending them to the left and right child nodes, respectively). The result is an even distribution of tokens at the leaf nodes. Diffracting trees have been introduced for *counting-problems*, and hence the leaf nodes are counters, assigning numbers to each token that exits from them. Moreover, the number of tokens that are output at the leaves, satisfy the *step property*, which states that: when there are no tokens present inside the tree and if out_i denotes the number of tokens that have been output at leaf i, $0 \leq out_i - out_j \leq 1$ for any pair i and j of leaf-nodes such that $i < j$ (i.e. if one makes a drawing of the tokens that have exited from each counter as a stack of boxes, the combined outcome will have the shape of a single step).

The fixed-size diffracting tree is optimal only for a small range of contention levels. To solve this problem, Della-Libera and Shavit proposed the *reactive diffracting trees*, where each node can shrink (to a counter) or grow (to a subtree with counters as leaves) according to the current load, in order to attain optimal performance [2]. The algorithm in [2] uses a set of parameters to make its decisions, namely folding/unfolding thresholds and the time-intervals for consecutive reaction checks. The parameter values depend on the multiprocessor system in use, the applications using the data structure and, in a multiprogramming environment, on the system utilization by the other programs that run concurrently. The programmer has to fix these parameters manually, using experimentation and information that is commonly not easily available (future load characteristics). A second characteristic of this scheme is that the reactive part is allowed to shrink or expand the tree only one level at a time, making the cost of a multi-adjustment phase on a reactive tree become high.

In this work we show that reactiveness and these two characteristics are not tied together: in particular, we present a tree-type distributed data structure that has the same semantics as the reactive trees that can expand or shrink many levels at a time, without need for manual tuning. To circumvent the need for manually setting parameters, we have analyzed the problem of balancing the trade-off between the two key measures, namely the contention level and the depth of the tree, in a way that enabled the use of efficient on-line methods for its solution. The new data structure is also considerably faster than the reactive diffracting trees, because of the low-overhead, multilevel reaction part: the new reactive trees can shrink and expand many levels at a time without using clock readings. The self-tuning reactive trees[1], like the reactive diffracting trees, are aimed in general for applications where such distributed data structures are needed. Since the latter were introduced in the context of counting problems, we use similar terms in our description, for reasons of consistency.

The rest of this paper is organized as follows. Section 2 presents the key idea and the algorithm of the self-tuning reactive tree. Section 3 describes the implementation of the tree. Section 4 presents an experimental evaluation of the self-tuning reactive trees, compared with the reactive diffracting trees, on the Origin2000 platform, and elaborate on a number of properties of our algorithm. Section 5 concludes this paper. Due to the space constraint, the correctness proof of our algorithm is presented in [3].

2 Self-tuning reactive trees

2.1 Problem description

The problem we are interested in is to construct a tree that satisfies the following requirements:

[1] We do not use term *diffracting* in the title of this paper since our algorithmic implementation does not use the *prism* construct, which is in the core of the algorithmic design of the (reactive) diffracting trees.

1. It must evenly distribute a set of concurrent process accesses to many small groups locally accessing shared data (counters at leaves), in a coordinated manner like the (reactive) diffracting trees. The step-property must be guaranteed.
2. Moreover, it must automatically and efficiently adjust its size according to its load in order to gain performance. It must not require any manually tuning parameters.

In order to satisfy these requirements, we have to tackle the following algorithmic problems:

1. Design a dynamic mechanism that would allow the tree to predict when and how much it should resize in order to obtain good performance whereas the load on it changes unpredictably. Moreover, the overhead that this mechanism will introduce should not exceed the performance benefits that the dynamic behavior itself will bring.
2. This dynamic mechanism should not only adjust the size of the tree in order to improve performance, but, more significantly, adjust it in a way that the tree still guarantees the fundamental properties of the structure, such as the step property.

2.2 Key idea

The ideal reactive tree is the one in which each leaf is accessed by only one process(or) –holding a token [2] – at a time and the cost to traverse it from the root to the leaves is kept minimal. However, these two latency-related factors are opposite to each other, i.e. if we want to decrease the contention at the leaves, we need to expand the tree and so the cost to traverse from the root to the leaves increases.

What we are looking for is a tree where the *overall overhead*, including the *latency due to contention* at the leaves and the *latency due to traversal* from the root to the leaves, is minimal and with *no manual tuning*. In addition to this, an algorithm that can achieve the above, must also be able to cope with the following difficulties: If the tree expands immediately when the contention level increases, then it will pay the expensive cost for travel and this cost is going to be unnecessary if after that the contention level suddenly decreases. On the other hand, if the tree does not expand in time when the contention-level increases, it has to pay the large cost of contention. If the algorithm knew in advance about the changes of contention-levels at the leaves in the whole time-period that the tree operates, it could adjust the tree-size at each time-point in a way such that the overall overhead is minimized. As the contention-levels change unpredictably, there is no way for the algorithm to know this kind of information, i.e. the information about the future.

To overcome this problem, we have designed a reactive algorithm based on the online techniques that are used to solve the online currency trading problem [4].

Definition 1. *Let* surplus *denote the number of processors that exceeds the number of leaves of the self-tuning reactive tree, i.e. the subtraction of the number of*

[2] For reasons of brevity, throughout the paper, instead of using the phrase "process(or) holding a token" we use simply the term process or processor

the leaves from the maximal number of processors in the system that potentially want to access the tree. The surplus *represents the contention level on the tree because the surplus processors cause contention on the leaves.*

Definition 2. *Let* latency *denote the latency due to traversal from the root to the leaves.*

Our challenge is to balance the trade-off between *surplus* and *latency*. Our solution for the problem is based on an optimal competitive algorithm called *threat-based algorithm* [4]. The algorithm is an optimal solution for the one-way trading problem, where the player has to decide whether to accept the current exchange rate as well as how many of his/her dollars should be exchanged to yens at the current exchange rate without knowledge on how the exchange rate will vary in the future.

2.3 The new algorithm

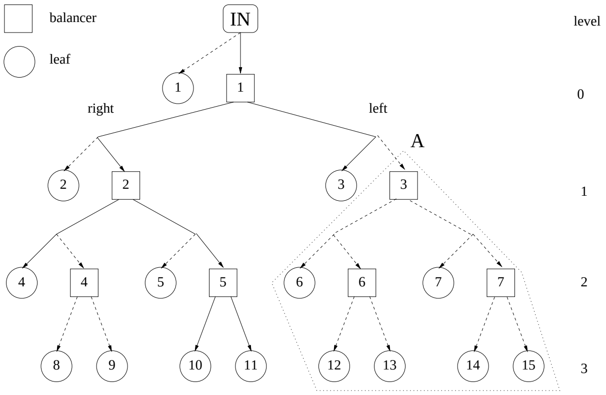

Fig. 1. A self-tuning reactive tree

In the self-tuning reactive trees, to adapt to the changes of the contention efficiently, a leaf should be free to shrink or grow to any level suggested by the reactive scheme in one adjustment step. With this in mind, we designed a data structure for the trees such that the time used for the adjustment and the time in which other processors are blocked by the adjustment are kept minimal. Figure 1 illustrates the self-tuning reactive tree data structure. Each balancer has a *matching* leaf with corresponding identity. Symmetrically, each leaf that is not at the lowest level of the tree has a *matching* balancer with corresponding identity. The squares in the figure are balancers and the circles are leaves. The numbers in the squares and circles are their identities. Each balancer has two outputs, $left$ and $right$, each of them being a pointer that can point to either a leaf or a balancer. A shrink or expand operation is essentially a switch of such a pointer (from the balancer to the matching leaf or from the leaf to the matching balancer, respectively). The solid arrows in the figure represent the present pointer contents.

Assume the tree has the shape as in Figure 1, where the solid arrows are the pointers' current contents. A processor p_i first visits the tree at its root IN, then following the root pointer visits 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 vise-versa) and then continues visiting the next node according to the toggle-bit. When visiting a leaf L, p_i before taking an appropriate counter value and exiting, checks the *reaction condition* according to the current load at L. The reaction condition estimates which tree level is the best for the current load.

The reaction procedure In order to balance the trade-off between *surplus* and *latency*, the procedure can be described as a game, which evolves in *load-rising* and *load-dropping transaction phases.*

Definition 3. *A* load-rising (resp. load-dropping) transaction phase *is a maximal sequence of subsequent visits at a leaf-node with monotonic non-decreasing (resp. non-increasing) estimated contention-level over the entire tree. A load-rising phase ends when a decrease in contention is observed; at that point a load-dropping phase begins.*

During a load-rising phase, a processor traversing that leaf may decide to expand the leaf to a subtree of depth that depends on the amount of the rising contention-level. That value is computed using the *threat-based on-line* method of [4], following the principle: “expand *just enough* to guarantee a bounded competitive ratio, even in the case that contention may drop to minimum at the next measurement”. Symmetric is the case during a load-dropping phase, where the reaction is to shrink a subtree to the appropriate level, depending on the measurement. The computation of the level to shrink to or to expand to uses the number of processors in the system as an upper bound of contention. The reaction procedure is described in detail in Section 3.2.

Depending on the result of checking the reaction condition, the processor acts as follows:

Recommended reaction: Grow to level l_{lower}, i.e. the current load is too high for the leaf L and L should expand to level l_{lower}. The processor, before exiting the tree through L, must help in carrying out the expansion task. To do so, the corresponding subtree must be constructed (if it was not already existent), the subtree's counters' (leaves') values must be set, and the pointer pointing to L must switch to point to its corresponding balancer, which is the root of the subtree resulting from the expansion.

Recommended reaction: Shrink to level l_{higher}, the current load at the leaf L is too low and thus L would like to cause a shrink operation to a higher level l_{higher}, in order to reduce the latency of traversing from the root to the present level. This means that the pointer to the corresponding balancer (i.e. ancestor of L) at level l_{higher} must switch to point to the matching counter (leaf) and the value of that counter must be set appropriately. Let B denote that balancer. The sub-tree with B as a root contains more leaves than just L, which might not have decided to shrink to l_{higher}, and thus the processor must take this into account. To enable processors do this check, the algorithm uses an asynchronous vote-collecting scheme: when a leaf L decides to shrink to level l_{higher}, it adds its *weighted vote* for that shrinkage to a corresponding vote-array at balancer B.

Definition 4. *The weight of the vote of leaf L is the number of lowest-level leaves in the subtree rooted at the balancer matching L.*

As an example in Figure 1 the weight of the vote of leaf 4 is 2. Note that when voting for balancer B, the leaf L is not concerned about whether B has shrunk into its matching leaf or not. The processor that helps L write its vote to B's vote-array, will then check whether there are enough votes collected at B's vote-array. If there are enough votes collected at B' vote-array, i.e. if the sum of their weights is more than half of the total possible weight of the sub-tree rooted at B (i.e. if more than half of that subtree wants to shrink to the leaf matching B), the shrinkage will happen. After completing the shrinkage task, the processor increases and returns the counter value of L, thus exiting the tree. In the checking process, the processor will abort if the balancer B has shrunk already by a concurrent operation.

In the shrinkage procedure, the leaf matching B and the leaves of the subtree rooted at B must be locked in order to (i) collect their counters' values, (ii) compute the next counter value for the leaf matching B and (iii) switch the pointer from B to its matching leaf. Note that all the leaves of subtree B need to be locked *only if* the load on the subtree is *so small that it should be shrunk to a leaf.* Therefore, locking the subtree in this case effectively behaves as if locking a leaf (i.e. as it is done in the classical reactive diffracting trees) from the performance point of view.

Example of executing grow: Consider a processor p_i visiting leaf 3 in Figure 1, and let the result of the check be that the leaf should grow to sub-tree A with leaves $12, 13, 14$ and 15: The processor first constructs the sub-tree, whereas at the same time other processors may continue to access leaf 3 to get the counter values and then exit the tree without any disturbance. After that, it locks leaf 3 in order to (i) switch the pointer to balancer 3 and (ii) assign the proper values to counters $12, 13, 14$ and 15, then it releases leaf 3. At this point, the new processors following the left pointer of balancer 1 will traverse through the new sub-tree, whereas the old processors that were directed to leaf 3 before, will continue to access leaf 3's counter and exit the tree. After completing the expansion task, p_i continues its normal task to access leaf 3's counter and exits the tree.

Example of executing shrink: Consider a processor p_i visiting leaf 10 in Figure 1 and let the result of the reaction condition be that the subtree should shrink to leaf 2. Because the sub-tree rooted at balancer 2 contains more leaves besides 10, which might not have decided to shrink to 2, processor p_i will check the votes collected at 2 for shrinking to that level. Assume that leaf 4 has voted for balancer 2, too. The weight of leaf 4's vote is two because the vote represents leaves 8 and 9 at the lowest level. Leaf 10's vote has weight 1. Therefore, the sum of the weights of the votes collected at balancer 2 is 3. In this case, processor p_i will help balancer 2 to perform the shrinkage task because the weight of votes, 3, is more than half of the total possible weight of the sub-tree (i.e. more than half of 4, which is the number of the leaves at the lowest level of the subtree –$8, 9, 10$ and 11). Then p_i locks leaf 2 and all the leaves of the sub-tree rooted at balancer 2, collects the counter values at them, computes the next counter value for leaf 2 and switches the pointer from balancer 2 to leaf 2. After that, all the leaves of the sub-tree are released immediately so that other processors can continue to access their counters. As soon as the counter at leaf 2 is assigned the new value, the new processors going along the right pointer of balancer 1 can access the counter and exit the tree whereas the old processors are traversing in the old sub-tree. After completing the shrinkage task, the processor exits the tree, returning the value from counter 10.

Space needs of the algorithm In a system with n processors, the algorithm needs $n - 1$ balancer nodes and $2n - 1$ leaf nodes. Note that it may seem that the data structure for the self-tuning reactive trees uses more memory space than the data structure for the reactive diffracting trees, since it introduces an auxiliary node (matching leaf) for each balancer of the tree. However, this is actually splitting the functionality of a node in the reactive diffracting trees into two components, one that is enabled when the node plays the role of a balancer and another that is enabled when the node plays the role of a leaf (cf. also Section 3.3 and Section 3.4). In other words, the corresponding memory requirements are similar. From the structure point of view, splitting the node functionality is a fundamental difference between the self-tuning trees and the reactive diffracting trees. The voting arrays' space needs at each balancer are $O(k)$, which are similar to the space needs for the prism at each balancer of the reactive diffracting trees, where k is the number of leaves of the subtree rooted at the balancer.

3 Implementation

3.1 Preliminaries

Data structure and shared variables: Figure 3 describes the tree data structure and the shared variables used in the implementation.

The synchronization primitives used for the implementation are *test-and-set (TAS)*, *fetch-and-xor (FAX)* and *compare-and-swap (CAS)*. Their semantics are described in [3]. Moreover, in order to simplify the presentation and implementation of our algorithm, we define, implement and use advanced synchronization operations: *read-and-follow-link* and *conditionally-acquire-lock*. The read-and-follow-link operations and the conditionally-acquire-lock operation are outlined in pseudo-code in Fig. 2. The way these locking mechanisms interact and ensure safety and liveness in our data structure accesses is explained in the descriptions of the implementations of the *Grow* and *Shrink* procedures and is proven in [3].

3.2 Reaction conditions

As mentioned in section 2.3, each leaf L of the self-tuning reactive tree estimates which level is the best for the current load. The leaf estimates the total load of tree by using the following formula:

```
$NodeType$ ASSIGN($NodeType * trace_i$, $NodeType * child$)
A0   $*trace_i := child$;/*mark $trace_i$ under update,clearing mask-bit*/
A1   $temp := *child$; /*get the expected value*/
A2   $temp.mask := 1$; /*set the mask-bit*/
A3   if ($local := CAS(trace_i, child, temp)) = child$ then return $temp$;
A4   else return $local$;

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

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

Fig. 2. The read-and-follow-link operations (Assign/Read) and conditionally-acquire-lock operation (AcquireLock_cond).

$$TotLoadEst = L.contention * 2^{L.level}$$

line C0 in *CheckCondition*() in Figure 3, where $MaxProcs$ is the maximum number of processors potentially wanting to access the tree and $L.contention$, the contention of a leaf, is the number of processors that currently visit the leaf. $L.contention$ is increased by one every-time a processor visits the leaf L and is decreased by one when a processor leaves the leaf. Because the number of processors accessing the tree cannot be greater than $MaxProcs$ we have an upper bound for the load: $TotLoadEst \leq MaxProcs$.

At the beginning, the initial tree is just a leaf, so the the initial *surplus*, *baseSurplus*, is $MaxProcs - 1$ and the initial *latency*, *baseLatency*, is 0. Then, based on the contention variation on each leaf, the values of *surplus* and *latency* are updated according to the online trading algorithm. Procedure *Surplus*2*Latency*() (respectively *Latency*2*Surplus*()) is invoked (lines C4, C5) to adjust the number of surplus processors that the tree should have at that time. The surplus value will be used to compute the number of leaves the tree should have and consequently the level the leaf L should shrink/grow to. .

Procedure $Surplus2Latency(L, TotLoadEst, FirstInPhase)$ in Figure 3 exchanges $L.surplus$ to $L.latency$ according to the *threat-based algorithm* [4] using $TotLoadEst$ as exchange rate. For self-containment, the computation implied by this algorithm is explained below. In a load-rising transaction phase, the following rules must be followed:

1. The tree is expanded only when the estimated current total load is the highest so far in the present transaction phase.
2. When expanding, expand *just enough* to keep the competitive ratio $c = \varphi - \frac{\varphi-1}{\varphi^{1/(\varphi-1)}}$, where $\varphi = \frac{MaxProcs}{2}$, even if the total load drops to the minimum possible in the next measurement.

Following these, the number of leaves the tree should have more is:

$$deltaSurplus = baseSurplus * \frac{1}{C} * \frac{TotLoadEst - TotLoadEst^{-}}{TotLoadEst - 2}$$

where $TotLoadEst^{-}$ is the highest observed total load before the present measurement and *baseSurplus* is the number of surplus processors at the beginning of the present transaction phase (line SL3, where mXY is the lower bound of the estimated total load). Everytime a new transaction phase starts, the value *baseSurplus* is set to the last value of *surplus* in the previous transaction phase (line C3). The parameter $FirstInPhase$ is used to identify whether this is the first exchange of the transaction phase. At the beginning,

$$surplus = baseSurplus = MaxProcs - 1$$

i.e. the tree degenerates to a node. Both variables $TotLoadEst^{-}$ and *baseSurplus* are stored in fields $TotLoadEst$ and *baseSurplus* of the leaf data structure, respectively.

Symmetrically, when the tree should shrink to reduce the traversal latency, the exchange rate is the inverse of the total load, $rXY = \frac{1}{TotLoadEst}$, which is increasing. In this case, the value of *surplus* increases and that of *latency* decreases.

3.3 Expanding a leaf to a sub-tree

A grow operation of a leaf L to a subtree T, whose root is L's matching balancer B and whose depth is $L.SugLevel - L.level$, essentially needs to (i) set the counters at the new leaves in T to proper values to ensure the step property; (ii) switch the corresponding child pointer of L's parent from L to B; and (iii) activate the nodes in T. (Figure 5 illustrates the steps taken in procedure grow, which is given in pseudocode in Figure 4.) Towards (i), it needs to:

- make sure there are no pending tokens in T. If there are any, *Grow* aborts (step G1 in *Grow*), since it should not cause “old” tokens get “new” values (that would cause “holes” in the sequence of numbers received by all tokens in the end). A new grow operation will be activated anyway by subsequent tokens visiting L, since L has high contention.
- acquire the locks for the new leaves, to be able to assign proper counter values to them (step G3 in *Grow*) to ensure the step property.
- make a consistent measurement of the number of pending processors in L and $L.count$ to use in the computation of the aforementioned values for the counters. Consistency is ensured by acquiring L's lock (step G4) and by switching L's parent's pointer from L to B (i.e. performing action (ii) described above; step G5 in *Grow*), since the latter leaves a “non-interfered” set of processors in L.

Each of these locks' acquisition is *conditional*, i.e. if some ancestor of L holds it, the attempt to lock will return fail. In such a case the grow procedure aborts, since the failure to get the lock means that there is an overlapping shrink operation by an ancestor of L. (Note that overlapping grow operations by an ancestor of L would have aborted, due to the existence of the token (processor) at L (step

```
type NodeType = record Nid : [1..MaxNodeId]; kind : {BALANCER, LEAF}; mask: bit; end;
     BalancerType = record state : {ACTIVE, OLD}; level : int; toggleBit : boolean;
          parent : [1..MaxNodeId]; leftChild, rightChild : NodeType;
          votes : array[1..SizeOfMySubtree] of int; end;
     LeafType = record state : {ACTIVE, OLD}; level, count, init : int;
          parent : [1..MaxNodeId]; lock : {0..MaxNodeId}; contention,totLoadEst : int;
          transPhase : {RISING, DROPPING};
          latency, baseLatency, surplus, baseSurplus, oldSugLevel, sugLevel : 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..MaxProcs] of [1..MaxNodeId];
private variables
     MyPath : array[1..MaxLevel] of NodeType; /*one for each processor*/

int CheckCondition(LeafType L)
C0   TotLoadEst := MIN(MaxProcs, L.contention * $2^{L.level}$);
C1   FirstInPhase := False;
C2   if (L.transPhase = RISING) and (TotLoadEst < L.totLoadEst) then
       L.transPhase := DROPPING; L.baseLatency := L.latency; FirstInPhase := True;
C3   else if (L.transPhase = DROPPING) and (TotLoadEst > L.totLoadEst) then
       L.transPhase := RISING; L.baseSurplus := L.surplus; FirstInPhase := True;
C4   if L.transPhase = RISING then Surplus2Latency(L, TotLoadEst, FirstInPhase);
C5   else Latency2Surplus(L, $\frac{1}{TotLoadEst}$, FirstInPhase);
     L.totLoadEst := TotLoadEst; L.oldSugLevel := L.sugLevel;
C6   L.sugLevel := $log_2$(MaxProcs − L.surplus);
     if L.sugLevel < L.level then return SHRINK;
     else if L.sugLevel > L.level then return GROW;
     else return NONE;

Surplus2Latency(L, TotLoadEst, FirstInPhase)
SL0  X := L.surplus; baseX := L.baseSurplus; Y := L.latency;
SL1  rXY := TotLoadEst; LrXY := L.totLoadEst;
SL2  if FirstInPhase then
        if rXY > mXY * C then deltaX := baseX * $\frac{1}{C}$ * $\frac{rXY-mXY*C}{rXY-mXY}$; /*C: comp. ratio*/
SL3  else deltaX := baseX * $\frac{1}{C}$ * $\frac{rXY-LrXY}{rXY-mXY}$;
SL4  L.surplus := L.surplus − deltaX; L.latency := L.latency + deltaX * rXY;

Latency2Surplus(L, $\frac{1}{TotLoadEst}$, FirstInPhase)
/* symmetric to the above with: X := L.latency; baseX := L.baseLatency; Y := L.surplus;
     rXY := $\frac{1}{TotLoadEst}$; LrXY := $\frac{1}{L.totLoadEst}$;*/
```

Fig. 3. The tree data structure and CheckCondition, Surplus2Latency and Latency2Surplus procedures

G1 in *Grow*).) Furthermore, the new leaves' locks are requested in *decreasing* order of node-id, followed by the request of *L.lock*, to avoid deadlocks.

Towards action (iii) from above, the grow procedure needs to reset the tree's T balancers' toggle bits and vote arrays (before switching L's parent's pointer from L to B; step G2) and set the state values of all balancers and bottom-level leaves in T to ACTIVE (after having made sure that the growing will not abort; step G9-G10).

3.4 Shrinking a sub-tree to a leaf

Towards a decision of whether and where to shrink to, the token at a leaf L_0 with recommended reaction to shrink to level $L_0.SugLevel$ must add L_0's vote in the vote arrays of the balancers of its path from the root, starting from level

```
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]) /* Can't miss any processors sincethe current ones go to Leaves[Nid]*/
         if ∃ pending processors in the subtree rooted at B then return; /*abort*/
G2   for each balancer B' in the subtree rooted at B,up to level L.sugLevel − 1
         forall entries i : B'.votes[i] := 0; B'.toggleBit = 0;
G3   for each leaf L' at level L.sugLevel of the subtree rooted at B,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
     /*$1^{st}$: an ancestor activated an overlapping Shrink; $2^{nd}$: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]) /*Can't miss any since the new ones go to B*/
         ppL := #(pending processors at L);
G7   CurCount := L.count; L.state := OLD;
G8   Release(L.lock);
G9   for each balancer B' as described in step G2 do B'.state := ACTIVE;
G10  for each leaf L' as described in step G3 do
         update L'.count using ppL and 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.oldSugLevel < L.sugLevel then /*new suggested level islower than older suggestion*/
E1      for(i := L.oldSugLevel; i < L.sugLevel; i + +) do Balancers[MyPath[i].Nid].votes[Nid] := 0;
     else for (i := L.sugLevel; i < L.oldSugLevel; i + +) do
E2          B := Balancers[MyPath[i].Nid];
E3          $B.votes[Nid] := 2^{MaxLevel-L.level}$; $bWeight := 2^{MaxLevel-B.level}$; /*weight of B's subtree*/
E4          if $\frac{\sum_i B.votes[i]}{bWeight} > 0.5$ 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   if (TAS(TokenToReact[Nid]) = 1) then return; /*abort, someone is doing the shrinkage*/
S2   forall i : Read(Tracing[i]) /*can't miss any since the currentones go to B*/
         if ∃ pending processor at L then return;/*abort*/
S3   if ( not AcquiredLock_cond(L.lock, Nid)) or (B.state = OLD) then
         /*$1^{st}$: some ancestor is performing Shrink; $2^{nd}$: someone already made the shrinkage*/
         Release possibly acquired lock; return; /*abort*/
S4   L.state := OLD; /*avoid reactive adjustment at L*/
S5   forall leaf L' in B's subtree, in increasing order of nodeId do
         AcquireLock_cond(L'.lock, Nid); /*No fails expected since Grow operations by ancestors
                                          will abort at G1*/
S6   Switch the parent's pointer from B to L
S7   forall i : Read(Tracing[i]); eppB := #(effective pending processors in B's subtree;
         /*can't miss any since the new ones go to L*/
S8   for each balancer B' in the subtree rooted at B do B'.state := OLD;
     SL := ∅; SLCount := ∅;
S9   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);
S10  L.count := f(eppB, SL, SLCount);
S11  L.state := ACTIVE;
S12  Release(L.lock);
S13  Reset(TokenToReact[Nid]);
```

Fig. 4. The Grow, Elect2Shrink and Shrink procedures

$L_0.SugLevel$, up to level $L_0.level - 1$ (it must also take care to remove potentially existing older votes at layers above that; step E1 in *Elect2Shrink* in Figure 4). When a balancer with enough votes is reached, the shrink operation will start

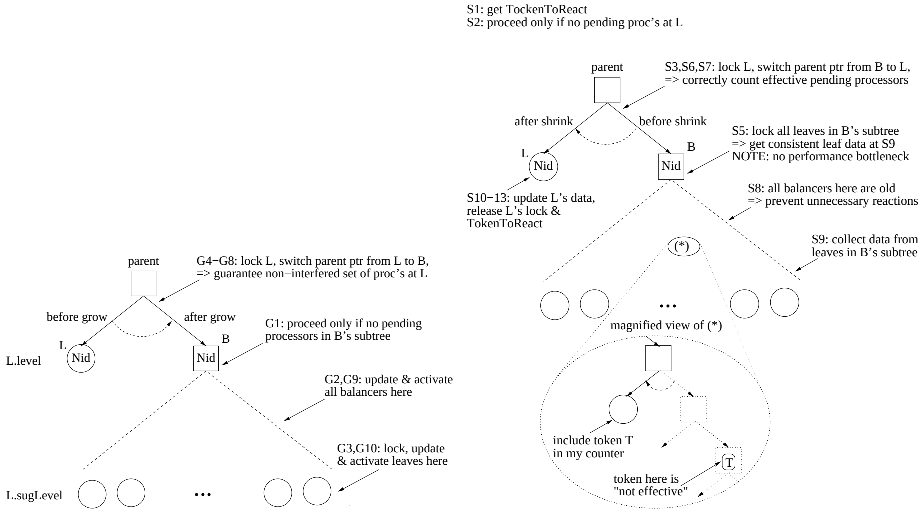
S1: get TockenToReact
S2: proceed only if no pending proc's at L
parent
S3,S6,S7: lock L, switch parent ptr from B to L,
=> correctly count effective pending processors
after shrink
before shrink
S5: lock all leaves in B's subtree
=> get consistent leaf data at S9
NOTE: no performance bottleneck
L
Nid
B
Nid
S10−13: update L's data,
release L's lock &
TokenToReact
S8: all balancers here are old
=> prevent unnecessary reactions
(*)
S9: collect data from
leaves in B's subtree
...
magnified view of (*)
include token T
in my counter
T
token here is
"not effective"
parent
G4−G8: lock L, switch parent ptr from L to B,
=> guarantee non−interfered set of proc's at L
before grow
after grow
G1: proceed only if no pending
processors in B's subtree
L
Nid
B
Nid
L.level
G2,G9: update & activate
all balancers here
G3,G10: lock, update
& activate leaves here
L.sugLevel
...

Fig. 5. Illustration for Grow and Shrink procedures

(steps E3-E4 in *Elect2Shrink*). Figure 5 and Figure 4 illustrate and give the pseudocode of the steps taken towards shrinking.

Symmetrically to a grow operation, a shrink from a subtree T rooted at balancer B (with enough votes) to B's matching leaf L, essentially needs to (i) set the counter at L to the proper value to ensure the step property; (ii) switch the corresponding child pointer of B's parent from B to L; and (iii) de-activate the nodes in T. Towards (i), it needs to:

- make sure there are no pending tokens in L. If there are any, shrink aborts (step S2 in *Shrink*), since it should not cause "old" tokens get "new" values. Subsequent tokens' checking of the reaction condition may reinitiate the shrinking later on anyway.
- acquire L's lock (step S3), to be able to assign an appropriate counter value to it, to ensure the step property.
- make a consistent measurement of (1) the number of pending processors in T and (2) the values of counters of each leaf L' in T. Consistency is ensured by acquiring $L'.lock$ for all L' in T (step S5) and by switching B's parent's pointer from B to L (i.e. performing action (ii) described above; step S6 in *Shrink*), since the latter leaves a "non-interfered" set of processors in T.

Similarly to procedure grow, these locks' acquisition is conditional. Symmetrically with grow, the requests are made first to $L.lock$ and then to the locks of the leaves in T, in *increasing* order of node-id, to avoid deadlocks. Failure to get $L.lock$ implies an overlapping shrink operation by an ancestor of L. Note that overlapping grow operations by an ancestor of L would have aborted, due to the existence of the token at B (step G1 in *Grow*). Note also that an overlapping shrink by some of L's ancestors cannot cause any of the attempts to get some $L'.lock$ to fail, since that shrink operation would have to first acquire the lock for L (and if it had succeeded in getting that, it would have caused the shrink from B to L to abort earlier, at step S3 of *Shrink*()).

Towards action (iii) from above, the shrink procedure sets the balancers' and leaves' states in T to OLD (steps S8-S9 in *Shrink*), after having made sure that the shrink will not abort.

4 Evaluation

In this section, we evaluate the performance of the self-tuning reactive trees proposed here. We used the reactive diffracting trees of [2] as a basis of comparison since they are the most efficient reactive counting constructions in the literature.

The source code of [2] is not publicly available and we implemented it following exactly the algorithm as it is presented in the paper. We used the full-contention benchmark, the index distribution benchmark [2] and the surge load benchmark [2] on the SGI Origin2000, a popular commercial ccNUMA multiprocessor.

In [2], besides running the benchmarks on a non-commercially availiable machine with 32 processors (Alewife), the authors also ran them on the simulator simulating a multiprocessor system similar to Alewife with up to 256 processors.

The most difficult issue in implementing the reactive diffracting 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 [2]. However, subsection *Load Surge Benchmark* in [2] described that the reactive diffracting tree sized to a depth 3 tree when they ran index-distribution benchmark [1] with 32 processors in the highest possible load ($work = 0$) and the number of consecutive timings was set at 10. According to the description, we run our implementation of the reactive diffracting tree on the ccNUMA Origin 2000 with 32 MIPS R10000 processors and the result is that folding and unfolding thresholds are 4 and 14 microseconds, respectively. This selection of parameters did not only keep our experiments consistent with the ones presented in [1] but also gave the best performance for the diffracting trees in our system. Regarding the prism size (prism is an algorithmic construct used in diffracting process in the reactive diffracting trees), each node has $c2^{(d-l)}$ prism locations, where $c = 0.5$, d is the average value of the reactive diffracting tree depths estimated by processors passing the tree and l is the level of the node [2, 5]. The upper bound for adaptive spin *MAXSPIN* is 128 as mentioned in [1].

In order to make the properties and the performance of the self-tuning reactive tree algorithm presented here accessible to other researchers and to help reproducibility of our results, C code for the tested algorithms is available at `http://www.cs.chalmers.se/∼phuong/sat_jul04.tar.gz`.

4.1 Full-contention and index distribution benchmarks

The system used for our experiments was a ccNUMA SGI Origin2000 with sixty four 195MHz MIPS R10000 CPUs with 4MB L2 cache each. The system ran IRIX 6.5. We ran the reactive diffracting tree *RD-tree* and the self-tuning reactive tree *ST-tree* in the full-contention benchmark, in which each thread continuously executed only the function to traverse the respective tree, and in the index

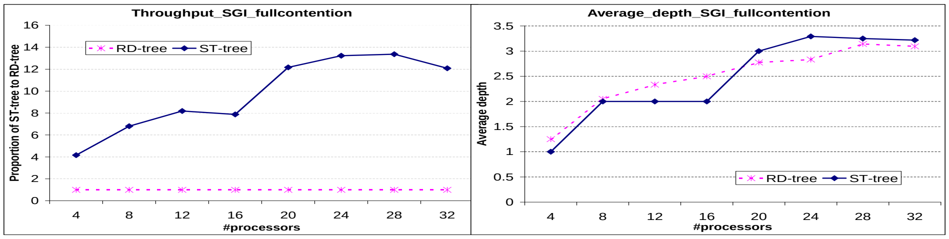

Fig. 6. Throughput and average depth of trees in the full-contention benchmark on SGI Origin2000.

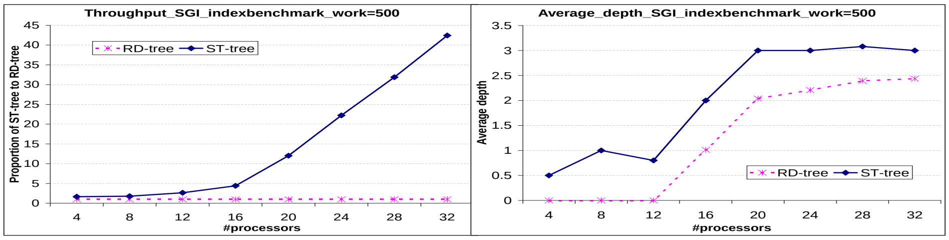

Fig. 7. Throughput and average depth of trees in the index distribution benchmark with $work = 500\mu s$ on SGI Origin2000.

distribution benchmark with $work = 500\mu s$ [2][1]. Each experiment ran for one minute and we counted the average number of operations per second.

Results: The results are shown in Figure 6 and Figure 7. The right charts in both the figures show the average depth of the ST-tree compared to the RD-tree. The left charts show the proportion of the ST-tree throughput to that of the RD-tree.

The most interesting result is that when the contention on the leaves increases, the ST-tree automatically adjusts its size close to that of the RD-tree that requires three experimental parameters for each specific system.

Regarding throughput and scalability, we observed that the ST-tree performs better than the RD-tree. This is because the ST-tree has a faster and more efficient reactive scheme. The surge load benchmark in the next subsection shows that the reactive trees *continuously* adjust their current size slightly around the average size corresponding to a certain load (cf. Figure 8). Therefore, an efficient adjustment procedure will significantly improve the performance of the trees.

Studing the figures closer, in the full-contention benchmark (Figure 6), we can observe the scalability properties of the ST-tree, which shows increased throughput with increasing number of processors (as expected using the aforementioned arguments) in the left chart. The right chart shows that the average depth of the ST-trees is nearly the same as that of the RD-tree, i.e. the reaction decisions are pretty close.

In the index distribution benchmark with $work = 500\mu s$, which provides a lower-load environment, the ST-tree can be observed to show very desirable scalability behavior as well, as shown in Figure 7. The charts of the average depths of both trees have approximately the same shapes again, but the ST-tree

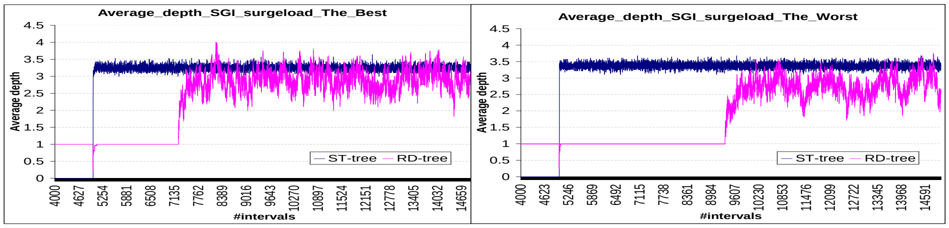

Fig. 8. Average depths of trees in the surge benchmark on SGI Origin2000, best and worst measurements. In a black-and-white printout, the darker line is the ST-tree.

expands from half to one depth unit more than RD-tree. This is because the throughput of the former was larger, hence the contention on the ST-tree leaves was higher than that on RD-tree leaves, and this made the ST-tree expand more.

4.2 Surge load benchmark

The benchmark shows how fast the trees react to contention variations. The benchmark is run on a smaller but faster machine[3], ccNUMA SGI2000 with thirty 250MHz MIPS R10000 CPUs with 4MB L2 cache each. On the machine the optimal folding and unfolding thresholds, which keep our experiments consistent with the ones presented in [1], are 3 and 10 microseconds, respectively. All other parameters are kept the same as the benchmarks discussed in the previous subsection.

In this benchmark we measured the average depth of each tree in each interval of 400 microseconds. The measurement was done by a monitor thread. At interval 5000, the number of threads was changed from four to twenty eight. The average depth of the trees at the interval 5001 was measured after synchronizing the monitor threads with all the new threads, i.e. the period between the end of interval 5000 and the beginning of interval 5001 was not 400 microseconds. Figure 8 shows the average depth of both trees from interval 4000 to interval 15000. The left chart shows the best reaction time figures for the RD-tree and the ST-tree; the right one shows the worst reaction time figures for the RD-tree and the ST-tree. In the benchmark, the ST-tree reached the suitable depth 3 for the case of 28 threads at interval 5004 in the best case and 5008 in the worst case, i.e. only after 5 to 8 intervals since the time all 28 threads started to run. The RD-tree reached level 3 at interval 7447 in the best case and at interval 9657 in the worst case. That means the reactive scheme introduced in this paper and used by the ST-tree makes the same decisions as the RD-tree, and, moreover, it reacts to contention variations much faster than the latter.

[3] This is because the first machine was replaced with that one at our computer center while this experimental evaluation was still in progress.

5 Conclusion

The self-tuning reactive trees presented in this work distribute the set of processors that are accessing them, to many smaller groups accessing disjoint critical sections in a coordinated manner. They collect information about the contention at the leaves (critical sections) and then they adjust themselves to attain adaptive performance. The self-tuning reactive trees extend a successful result in the area of reactive concurrent data structures, the reactive diffracting trees, in the following way:

- The reactive adjustment policy does not use parameters which have to be set manually and which depend on experimentation.
- The reactive adjustment policy is based on an efficient adaptive algorithmic scheme.
- They can expand or shrink many levels at a time with small overhead.
- Processors pass through the tree in only one direction, from the root to the leaves and are never forced to go back.

Moreover, the self-tuning reactive trees:

- have space needs comparable with that of the classical reactive diffracting trees
- exploit low contention cases on subtrees to make their locking process as efficient as in the classical reactive diffracting trees although the locking process locks more nodes at the same time.

Therefore, the self-tuning reactive trees can react quickly to changes of the contention levels, and at the same time offer a good latency to the processes traversing them and good scalability behavior. We have also presented an experimental evaluation of the new trees, on the SGI Origin2000, a well-known commercial ccNUMA multiprocessor. We think that it is of big interest to do a performance evaluation on modern multiprocessor systems that are widely used in practice.

Last, we would like to emphasize an important point. Although the new trees have better performance than the classical ones in the experimental evaluation conducted and presented here, this is not the main contribution of this paper. What we consider as main contribution is the ability of the new trees to self-tune their size efficiently without any need of manual tuning.

References

1. Shavit, N., Zemach, A.: Diffracting trees. ACM Trans. Comput. Syst. **14** (1996) 385–428
2. Della-Libera, G., Shavit, N.: Reactive diffracting trees. J. Parallel Distrib. Comput. **60** (2000) 853–890
3. Ha, P.H., Papatriantafilou, M., Tsigas, P.: Self-adjusting trees. Technical Report 2003-09, Computing Science, Chalmers University of Technology (2003) `http://www.cs.chalmers.se/~phuong/ SAT_TR.ps.gz`.
4. El-Yaniv, R., Fiat, A., Karp, R.M., Turpin, G.: Optimal search and one-way trading online algorithms. Algorithmica **30** (2001) 101–139
5. Shavit, N., Upfal, E., Zemach, A.: A steady state analysis of diffracting trees. Theory of Computing Systems **31** (1998) 403–423