Technical Report no. 2003-09

Self-Adjusting Trees

Phuong Hoai Ha, Marina Papatriantafilou, Philippas Tsigas

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, 2003

Technical Report in Computing Science at
Chalmers University of Technology and Göteborg University

Technical Report no. 2003-09
ISSN: 1650-3023

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, 2003

Abstract

The reactive diffracting trees *are known efficient distributed data structures for supporting synchronization. They not only distribute a set of processes to smaller groups accessing different parts of the memory in a global coordinated manner, but also adjust their size in order to attain efficient performance across different levels of contention. However, the existing reactive adjustment policy of these trees is sensitive to parameters that have to be manually set in an optimal way and be determined after experimentation. Because these parameters depend on the application as well as on the system configuration, determining their optimal values is hard in practice. Moreover, because the reactive diffracting trees expand or shrink one level at a time, the cost of a multi-adjustment phase on a reactive tree can become high. We argue that these two problems are not fundamental, and that it is possible to construct reactive trees that: (i) are self-adjustable with no need of fixing manually any parameter, and (ii) have the ability to expand or shrink many levels at one time.*

In this paper, we present a new distributed data structure called self-adjusting tree *that has the same specifications as the reactive diffracting trees but can expand or shrink many levels in any adjustment step. This feature helps the self-adjusting 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 the other processes traversing the tree at the same time. More significantly, the reactive policy of the self-adjusting tree is based on a scheme where both the contention of the tree and the adjustment decisions are computed with* no need *for the user to estimate any experimental parameter. The self-adjusting trees perform faster than the reactive diffracting trees on the SGI Origin2000, a well-known commercial ccNUMA multiprocessor.*

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 [5] are highly 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 global 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 (i.e. if one draws the tokens that have exited from each counter as a stack of boxes, the combined outcome will have the shape of a step).

The fixed-size diffracting tree is optimal for a small range of contention-levels. Della-Libera and Shavit proposed improved, *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 [1].

The algorithm of the reactive diffracting trees 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. A second characteristic of this scheme is that the reactive part is allowed to shrink or expand the tree one level at a time. This can become a performance bottleneck.

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 and that can expand or shrink many levels at a time, without need for manual tuning. To circumvent the need for manually set parameters, we have analyzed the problem of balancing the trade-off between two key measures, namely the contention level and the depth of the tree, in a way that enabled the use of efficient online 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 self-adjusting trees[1], like the reactive diffracting trees, are aimed in general for applications where such distributed

[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.

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. The self-adjusting tree and the design of the data structure that supports the tree are described in Section 2. The algorithm according to which the tree self-adjusts to the changes of contention-levels is presented in Section 3. Besides, the informal and the detailed descriptions of the algorithm (given in sections 2 and 3, respectively) include the main arguments for its correctness. The correctness proof of our algorithm is presented in Section 4. Section 5 presents an experimental evaluation of the self-adjusting tree, compared with the reactive diffracting tree, on the Origin2000. Section 6 concludes this paper.

2 The self-adjusting trees

The ideal reactive tree is the one in which each leaf is accessed by only one process(or) –or 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. 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, 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 expensive 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 online trading techniques [2]. The rest of this section gives an informal description of the algorithm, including a new data structure supporting the tree and examples of the reactive phases.

In the self-adjusting trees, to adapt to the changes of the contention efficiently, a leaf should be free to shrink or grow to any level in the tree suggested by the reactive scheme in

[2]The terms *processor, process* and *token* are used interchangeably throughout the paper

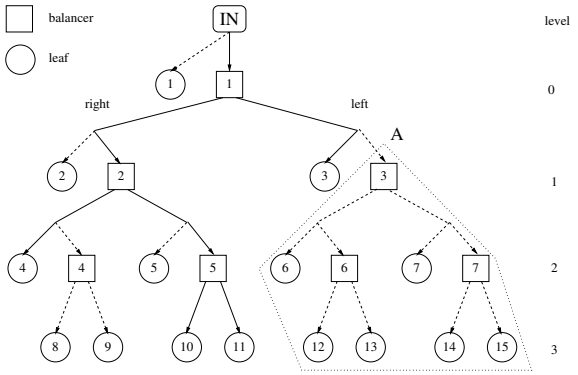

Figure 1. Self-adjusting tree

one adjustment step. With this in mind, we designed a *data structure* for the tree 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-adjusting 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 counter or from the counter to the matching balancer, respectively). The solid arrows in the figure represent the present pointer contents. Note that it may seem that the data structure for the self-adjusting tree 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 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. In other words, the corresponding memory requirements are comparable. Figure 2 describes the data structure of nodes in the tree.

Assume the tree has the shape as in Figure 1, where the solid arrows are the pointers' current contents. A processor called p_i first visits the tree at its root IN, then following the root pointer visits balancer 1. When visiting a balancer, it 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, the processor 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 exact formulation of the condition is given in the subsequent subsection 3.3. Depending on the result of the check,

```
struct node_t{
  state_t state;
  /*{ROOT, BALANCER, OLDBALANCER, LEAF, OLDLEAF}*/
  int id, level;
  lock_t rLock; /*reaction lock*/
  node_t* parent;
  /*If Balancer*/
  int toggle;
  node_t *left, *right;
  int votes[size];
  /*size: the number of descendent leaves*/
  /*If Leaf*/
  int init, count;
  lock_t nLock; /*node lock*/
  /*Part used for self-adjustment*/
  int np;
  /*the current number of processors at the leaf*/
  suggestion_t suggestion; /*{NONE, SHRINK, GROW}*/
  direction_t direction; /*{UPWARD, DOWNWARD}*/
  int osLevel, sLevel; /*[old] suggested level*/
  int latency,initLatency,surplus,initSurplus,rootL;
}
```

Figure 2. Node data structure

the processor acts as follows:

i) If the result is $None$, i.e. the current load is acceptable, the processor only increases the counter's value and exits the tree, returning this value.

ii) If the result is $Grow$ to level l_l, i.e. the current load is too high for the leaf L and L should expand to level l_l, the processor, before exiting the tree through L as in the above case, 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.

As an example, consider a processor visiting leaf 3 in Figure 1, and the result of the check is 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 accessing leaf 3's counter and exit the tree. After completing the expansion task, the processor continues its normal task to access leaf 3 counter and exits the tree.

iii) If the result is $Shrink$ to level l_h, 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_h 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_h must switch to point to the matching counter and the value of that counter must be set appropriately. Let B_h denote that balancer. The sub-tree with B_h as a root contains more leaves than just L, which might not have decided to shrink to l_h, 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_h, it adds its *weighted vote* for that shrinkage to a corresponding vote-array at balancer B_h. The weight of L's vote equals the number of lowest-level leaves in the subtree rooted at the balancer matching L. If there are enough votes collected at B_h' vote-array, i.e. if the sum of their weights are more than half of the total possible weight of the sub-tree rooted at B_h (i.e. if more than half of that subtree wants to shrink to the leaf matching B_h), the shrinkage will happen. To complete the shrinkage, the leaf matching B_h and all the leaves of the sub-tree rooted at B_h must be locked in order to (i) collect their counters' values, (ii) compute the next counter value for the leaf matching B_h and (iii) switch the pointer from B_h to its matching leaf. After that, all the leaves of the sub-tree are released immediately so that other processors can continue to access their counters. After completing the shrinkage task, the processor increases and returns the counter value of L, thus exiting the tree.

As an example, consider a processor p_i visiting leaf 10 in Figure 1 and 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 weights of votes 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. 2 which is half of 4, which would be the sum of the leaf weights if all leaves at the lowest level of the subtree –$8, 9, 10$ and 11– had voted for the shrinkage). Then the processor 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.

In the next sections, we give a more detailed description of the self-adjusting tree algorithm, including the reaction

condition and all the synchronization involved in the shrinkage and expansion phases.

3 Implementation

3.1 Preliminaries

Synchronization primitives: The synchronization primitives used for the implementation are *test-and-set (TAS)*, *fetch-and-xor (FAX)* and *compare-and-swap (CAS)*. The definitions of the primitives are described in Figure 3, where x is a variable and v, old, new are values.

```
TAS(x)                  FAX (x, v)              CAS (x, old, new)
/* init: x := 0 */       atomically{             atomically{
atomically{                 oldx := x;              oldx := x;
    oldx := x;              x := x xor v;           if (x = old) then
    x := 1;                 return oldx;               x := new;
    return oldx;            }                       return oldx;
    }                                               }
```

Figure 3. Synchronization primitives

In our implementation, we use two advanced operations called *two-word assignment* and *conditionally acquiring lock* operations, which are described in Figure 4

3.1.1 Two-word assignment operations

In our self-adjusting tree implementation, an array $Tracing$ is used to keep track where each processor is in the tree (Figure 5). Processor p_i will write to its corresponding element $Tracing[i]$ the address of the node it will go before it visits the node. Each element $Tracing[i]$ can be updated by only one processor p_i via procedure *Assign(pnode_t *trace_i, pnode_t *child)* and can be read by many other processors via procedure *Read(pnode_t *trace_i)*. The former will read the variable pointed by $child$ and then write its value to the variable pointed by $trace_i$. The latter will read the variable pointed by $trace_i$. They are the unique procedures that can access array $Tracing$. They are designed so that $Assign()$ is atomic to $Read()$

Lemma 3.1. Assign() *is atomic to* Read().

Proof. (Sketch) In procedure *Assign(pnode_t *trace_i, pnode_t *child)*, the variable pointed by $trace_i$ is first locked by writing the pointer $child$ to it (line 1). We exploit the last bit in the pointer, which is unused because of word-alignment memory architecture, to identify if the variable is locked or not. If the last bit of the variable $*trace_i$ is zero, the variable is locked and its value is the address of the other variable whose value must be written to $*trace_i$. After reading the expected pointer, $*child$, we set the last bit to 1 so as to unlock the variable $*trace_i$ and then write this value to the variable $*trace_i$ by *compare_and_swap* operations (lines 2-4).

In procedure $Read(pnode_t * trace_i)$, when reading the value of the variable $*trace_i$, we check if the variable contains the expected value (line 3). If the last bit of the variable is zero, the procedure will help procedure $Assign()$ to write the expected value to the variable before trying to read again (lines 4-6). The linearization point of procedure $Read()$ is the point the procedure reads the value with the non-zero last bit and the linearization point of procedure $Assign()$ is the point the procedure writes a zero-last-bit value to the variable $*trace_i$ (line 1 in $Assign()$). □

```
typedef node_t* pnode_t;

void Assign( pnode_t *trace_i, pnode_t *child) {
1  *trace_i = (pnode_t)child;   /* lock *trace_i */
2  temp = (int)*child; /* get the expected value */
3  temp += 1;                  /* Set the mask-bit */
4  CAS( (int*)trace_i, (int)child, temp);
}

pnode_t Read( pnode_t *trace_i) {
1  do
2    local = (int)*trace_i;
3    if( (local != 0) && (local & 1 == 0)) /* *trace_i is locked */
4      temp = *(int *)local;   /* help corresponding Assign() ...*/
5      temp += 1;
6      CAS( (int *)trace_i, local, temp);
7  while( local & 1 == 0);   /* ... until the Assign() completes */
8  return( (pnode_t)(local & MASK));       /* omit mask-bit */
}

bool Acquire_lock_c(lock_t *lock_p, int nId) {
 1  int b=BASE; old = *lock_p;
 2  if( (old > 0) && IsParent( old, nId))
 3    return FAIL;
 4  if( !CAS( lock_p, 0, nId))
 5    do
 6      for( i=b; i; i--);//delay
 7      b = min( b*FACTOR, CAP);
 8      old = *lock_p;
 9      if(old)
10        if( IsParent( old, nId)) return FAIL;
11        continue;
12    while( !CAS(lock_p, 0, nId));
13  return SUCCEED;
}
```

Figure 4. Assign, Read and Acquire_lock_c procedures

3.1.2 Conditionally acquiring lock operations

In both following expansion and shrinkage processes, the processor has to use procedure $Acquire_lock_c(\ell, nId)$ to

acquire the node-lock of a leaf. In the procedure $Acquire_lock_c(\ell, nId)$, the processor helps node n acquire a lock ℓ of a leaf by writing the node identity nId to the lock. If the lock node n requires was acquired by an ancestor of node n, the procedure returns $Fail$ (lines 3, 10). We implement the procedure by busy-wait with exponential back-off technique instead of queue-lock because the contention at the leaf of our self-adjusting tree is kept low. The tree will automatically expand to reduce contention on the leaves when the contention becomes high. In the low contention environment, the busy-wait with exponential back-off technique achieves better performance than does the queue-lock, which uses more complicated data structure [3]. All processors use the procedure to acquire locks for expansion and shrinkage tasks. For acquiring the node lock of a leaf to increase its counter value, i.e. not self-adjustment task, processors use procedure $Acquire_lock$ without condition. The procedure is similar to $Acquire_lock_c$ but does not check ancestor-condition (lines 3, 10). Procedure $Acquire_lock$ always returns $Succeed$.

In the next subsection, we will present in detail how a processor traverses the self-adjusting tree. The pseudo-code is described in Figure 5.

3.2 Traversing self-adjusting trees

Every processor i traverses the tree by calling function $Traverse_tree()$. Firstly, the processor visits *Root* (lines 2,3) and then goes to $Root$ counter if the tree degenerated into one counter or goes to $Root$ sub-tree whose root is a balancer. Before visiting *Root* balancer or *Root* counter, the processor pid updates its new location in $Tracing[pid]$ (line 8). The path along which the processor traverses the tree is recorded in variable $path$, where $path[i]$ is the balancer or counter the processor visited at level $i-1$. The array is used to keep track of which balancers the processor has passed by. According to the kinds of nodes the processor visits, it will behave correspondingly:

- *Balancer/Old balancer*[3]: the processor will call procedure $Balancer()$ to read and switch the balancer toggle-bit (line 11 in Traverse_tree()).
- *Old leaf*: the processor will call procedure $Leaf()$ to read and increase the counter $n \rightarrow count$ (line 13).
- *Leaf*: the processor, beside doing the same work as visiting an old leaf, will take part in the reactive process. The processor calls procedure $Check_reaction_condition()$ to check whether the current contention at the leaf is acceptable (line 20). If the result is $Shrink$, the processor will start the shrinkage procedure by calling $Elect2Shrink()$ (line 22). If the result is $Grow$, the processor will call procedure $Grow()$ to expand the leaf to a subtree (line 25).

[3]Intuitively, old balancers/old leaves are the nodes that a new processor going from the root can not visit at that time

```
node_t Balancers[MAX_BALANCERS];
node_t Leaf[MAX_LEAVES], *Root;
node_t *Tracing[NumPros];

int Traverse_tree( int pid){
 1  node_t *n;
 2  Assign( &Tracing[pid], &Root); /* record pid's current position */
 3  n = Read( &Tracing[pid]);
 4  for( i=0;; i++)
 5    path[i] = n;                                  /* trace the token */
 6    switch( n->state)
 7      case ROOT:
 8        Assign( &Tracing[pid], &n->left);
 9        n = Read( &Tracing[pid]);
10      case BALANCER or OLDBALANCER:
11        n = Balancer( n, pid);
12      case OLDLEAF:
13        result = Leaf( n, pid);
14        Assign( &Tracing[pid], &Root);        /* reset Tracing[pid] */
15        return( result);
16      case LEAF:
17        fetch&increment( &n->np); /*count current processors at n*/
18        if( TATAS( &n->rLock))       /* acquire the reaction-lock */
19          if(n->suggestion == NONE)  /* no pending reaction-task */
20            Check_reaction_condition(n);
21          if( n->suggestion == SHRINK)
22            if( Elect2Shrink( n, path))
23              n->suggestion = NONE;
24          else if( n->suggestion == GROW)
25            if( Grow(n))
26              n->suggestion = NONE;
27          release( n->rLock);
28        answer = Leaf( n, pid);
29        fetch&decrement( n->np);
30        Assign( &Tracing[pid], &Root);
31        return answer;
}

node_t *Balancer( node_t *n, int pid) {
1  k = fetch&xor( &n->toggle, 1);
2  if( k == 0)
3    Assign( &Tracing[pid], &n->right);
4  else
5    Assign( &Tracing[pid], &n->left);
6  return( Read( &Tracing[pid]));
}

int Leaf( node_t *n, int pid){
1  if( n->state in {LEAF, OLDLEAF})
2    acquire_lock( &n->nLock, n->id);                /* lock the leaf */
3    result = n->count;
4    n->count += pow( 2, n->level);
5    release_lock( &n->nLock);
6  return result;
}
```

Figure 5. Traverse_tree, Balancer and Leaf procedures

The next subsection will describe in more detail how the reaction condition is checked.

3.3 Reaction conditions

As mentioned in section 2, each leaf n of our self-adjusting tree estimates which level is the best for the current load by our reactive scheme. The leaf estimates the current load at the root of the tree by using the following formula:

$$load_{root} = load_n * 2^{level_n} \quad (1)$$

where $load_n$ is the current number of processors at leaf n and the tree is assumed balanced at level $level_n$ (line 1 in $Check_reaction_condition()$, where $NumPros$ is the maximum number of processors).

To apply the online trading algorithms on the self-adjusting tree, we first need to specify when the transaction starts and ends. Relying on the transaction specification, we can apply the algorithms in a proper way so that the competitive property holds.

Definition 3.1. *A new transaction of a leaf starts when:*

- *the first processor visits the leaf.*
- *the estimated load has been increasing/decreasing but now it starts to decrease/increase.*
- *the value of* surplus *is over the threshold* $(NumPros-1)$.

If the estimated load at the root increases continuously (lines 3,4), the tree should grow to have more leaves in order to reduce the contention at the leaves (line 5). That is, it will *reduce the processor surplus*, the number of processor exceeding the number of leaves, and *accept the traversal latency to increase*. To compute the exact adjustment measures, procedure $ExchangeX2Y$ is used.

Otherwise, if the estimated load has been increasing but now it starts to decrease, we say that a new *transaction* starts. The tree will shrink in order to *reduce the traversal latency* and will *accepts the load to increase* slightly (lines 6-9). To compute the exact adjustment measures, procedure $ExchangeX2Y$ is used again. The increase of the processor surplus has as effect the decrease of the number of leaves.

The case that the estimated load at the root decreases is similar to the case described above, but the roles of the surplus and the latency are switched to each other (lines 11-18). If the estimated load at the root decreases continuously, the tree should shrink to reduce the latency (line 13). Otherwise, if the load has decreased so far, but now starts to increase (line 14), again a new *transaction* starts. The tree will grow in order to reduce the surplus when the load on the leaves is high (lines 14-17).

```
Check_reaction_condition( node_t *n) {
 1  rootL = min( NumPros, n->np*pow(2, n->level));
 2  iRootL = 1/rootL;
 3  if ( n->direction == UPWARD) { /* the load has increased continuously*/
 4    if( rootL > n->rootL)                    /* the load continues increasing */
 5      ExchangeX2Y( surplus, latency, rootL, FALSE);
 6    else if ( rootL < n->rootL)              /* the load starts to decrease */
 7      n->direction = DOWNWARD;               /* start a new transaction ...*/
 8      n->initLatency = n->latency;
 9      ExchangeX2Y( latency, surplus, iRootL, TRUE);
10  }
11  else {            /* DOWNWARD - the load has decreased continuously*/
12    if( rootL < n->rootL)
13      ExchangeX2Y( latency, surplus, iRootL, FALSE);
14    else if ( rootL > n->rootL)
15      n->direction = UPWARD;
16      n->initSurplus = n->surplus;
17      ExchangeX2Y( surplus, latency, rootL, TRUE);
18  }
19  n->osLevel = n->sLevel;
20  n->sLevel = log(NumPros - n->surplus)/log(2);
21  if( n->osLevel != n->sLevel)
22    if( n->sLevel < n->level)
23      n->suggestion = SHRINK;
24    else if( n->sLevel > n->level)
25      n->suggestion = GROW;
}

ExchangeX2Y( X, Y, rXY, init) {
1  if( init)                                        /* the first exchange */
2    if( rXY > mXY*C)  /* C: competitive ratio; mXY: lower bound of rXY */
3      deltaX= n->initX*1/C*( rXY-mXY*C)/( rXY-mXY);
4  else
5    deltaX = n->initX*1/C*( rXY-n->rXY)/( rXY-mXY);
6  n->X -= deltaX;
7  n->Y += deltaX*rXY;
8  n->rXY = rXY;
}
```

Figure 6. Check_reaction_condition and ExchangeX2Y procedures

The outcome of the reaction condition whether to shrink, expand or do nothing, is drawn by computing the suggested level $sLevel_n$ for leaf n using the on-line adjustment result, as has been computed by procedure $ExchangeX2Y$. Relying on the difference between $sLevel_n$ and the current one, $level_n$, the leaf itself can know whether it should shrink to a higher-level leaf or grow to a lower-level one (lines 22-25).

Procedure $ExchangeX2Y(X, Y, rXY, init)$ exchanges X to Y with exchange rate from X to Y rXY according to the *thread-based algorithm* [2], where (X, Y) can be $(surplus, latency)$ or $(latency, surplus)$. When we expand the tree to reduce the load on the leaf, rXY is the estimated load at the root that is increasing (lines 5, 17 in $Check_reaction_condition()$), $rXY = rootL \in [1, NumPros]$.

In a transaction, the following rules must be followed:

1. The tree is expanded only when the current estimated load at the root is the highest so far.

2. When expanding, expand *just enough* to keep the competitive ratio $c = \varphi - \frac{\varphi-1}{\varphi^{1/(\varphi-1)}}$, where $\varphi = NumPros$, the maximum number of processors in the system.

The number of leaves the tree should have more is $deltaSurplus = initSurplus * 1/C * (rootL - rootL^{-})/(rootL - 1)$, where $rootL^{-}$ is the highest estimated load at the root before the present measurement and $initSurplus$ is the number of surplus processors at the beginning of the transaction (line 5).

Similarly, when the tree shrinks to reduce the traversal latency, the rXY is the inverse of the load at the root, $rXY = 1/rootL$, which is increasing. In this case, the value of $surplus$ increases and that of $latency$ decreases. When the value of $surplus$ is over the threshold $(NumPros - 1)$, the *current transaction ends and a new transaction starts* with $surplus = (NumPros - 1)$ and $latency = 0$. Parameter $init$ is used to identify whether this is the first exchange of the transaction.

In the two next subsections, we will present how the self-adjusting tree expands and shrinks. The pseudo-codes of the expansion and shrinkage processes are in Figure 7 and Figure 9.

```
bool Grow( node_t *n) {
 1  partner = &Balancers[n->id];
 2  for(i=0; i<NumPros; i++) /* check pending processors in the subtree*/
 3    trace_i = Read( &Tracing[i]);
 4    if( (trace_i == partner) || IsParent( n->id, trace_i->id))
 5      return FAIL;
 6  Reset all balancers in the partner subtree;
 7  Lock all leaves at level n->sLevel in the sub-tree
 8  in descreasing order of node Id;
 9  if( !acquire_lock_c(&n->nLock, n->id) || (n->state != LEAF))
10    Release all the locked leaves;
11    return FAIL;
12  Set all the locked leaves to the state OLDLEAF;
13  pr = n->parent;
14  branch = n->id % 2;
15  if( branch==0)
16    pr->right = partner;
17  else
18    pr->left = partner;
19  count = n->count;
20  np = 0;                          /*counting pending processors at n */
21  for( i=0; i<NumPros; i++)
22    if( Read( &Tracing[i]) == n) np++;
23  n->state = OLDLEAF;
24  release_lock(&n->nLock);
25  Assign counter values to the locked leaves,
26  set their state to LEAF and release them;
27  return SUCCEED;
}
```

Figure 7. Grow procedures

3.4 Expanding a leaf to a sub-tree

To grow a leaf n to a sub-tree whose root is the matching balancer of the leaf and whose depth is $(sLevel_n - level_n)$, the processor p_i doing the expansion task for n must ensure that there is no pending processor (old token) in that sub-tree (lines 2-5 in $Grow()$).

If there is no pending processor in the sub-tree, processor p_i tries to reset all balancers of the sub-tree (i.e. among other parts, set their toggle-bits to 0). It does not need to lock them because only processor p_i can access them. This is so, because there are no old tokens in the subtree, while newer tokens are directed to n, the leaf (counter) that currently wants to expand. New tokens will start coming into the grown subtree when the grow operation is complete, i.e. when the pointer to leaf n switches to point to n's matching balancer, the root of the subtree. The case that an ancestor of leaf n is expanding a leaf to the sub-tree containing leaf n never occurs because in the sub-tree there is at least a pending processor, p_i.

Then, processor p_i tries to lock all leaves of the sub-tree at level $sLevel_n$ in decreasing order of node identity and then reset them so that their $state$, np and $suggestion$ are set to $Oldleaf$, 0 and $None$, respectively and their self-adjustment variables are set to those of leaf n. If the locking process is unsuccessful because one of the leaves was locked by an *ancestor* of leaf n in the tree, all the leaves locked for leaf n are released and procedure $Grow()$ returns $Fail$ immediately. The reason is that if an ancestor of leaf n is shrinking the sub-tree containing n, it is worthless for leaf n to try to expand; instead, processor p_i should exit from n as fast as possible. The case that an ancestor of leaf n is expanding a leaf to the sub-tree containing leaf n never occurs as explained above.

Finally, after locking all necessary leaves of the sub-tree corresponding to leaf n, processor p_i tries to lock leaf n, which is being accessed by other processors (line 9). If processor p_i successfully locks the leaf n and the state of n is still $Leaf$, the processor switches the pointer from leaf n to the corresponding balancer by updating $right_{parent}$ (or $left_{parent}$) to $partner$ (lines 15-18). The number of pending processors in leaf n and the current counter value of leaf n are recorded in order to compute the successive counter values for the new leaves in the new sub-tree (lines 19-22). The state of leaf n is changed to $OldLeaf$ before leaf n is released so that no more reactive action occurs at leaf n (line 23). Each leaf in the new sub-tree will be released as soon as its variable $count$ is updated so that other processors can visit the leaf to get counter value as soon as possible.

In the paragraph, we describe how to assign the new counter values for the new leaves in $Grow()$ (line 25). Because all toggle-bits of the balancers in the new sub-tree are

reset to 0, i.e. the first processor passing through the sub-tree will arrive at the right-most leaf[4] at level $sLevel_n$. The code for assigning counter values for these new leaves is as in Figure 8, where $count$ and np are the values read at lines

```
next_count_value = count + np*pow(2,n->level);
increment = next_count_value - RightMostLeaf->init;
for( every leaf L at level n->sLevel)
  L->count = L->init + increment;
```

Figure 8. Assigning counter values for the new leaves

19 and 22 in $Grow()$. Local variable $next_count_value$ is the successive counter value after np processors leave leaf n.

3.5 Shrinking a sub-tree to a leaf

To shrink a sub-tree to its corresponding leaf, more than half of the leaves in the sub-tree need to vote for that leaf. Therefore, unlike the expansion case where a leaf n itself can expand to a sub-tree, in the shrinkage case a leaf n *only can vote* for the leaf ℓ it wants to shrink to, via ℓ's corresponding balancer. Only when the weight of all votes for the balancer is more than half of the weight of its sub-tree, the processor on behalf of n can perform the shrinkage task. The pseudo-code of the procedures used in the shrinkage process are described in Figure 9.

Procedure $Elect2Shrink()$ is executed in order to check whether a balancer satisfies the shrinkage condition after the procedure $Check_reaction_condition()$ was executed for a leaf n and the result was $n.suggestion = Shrink$. If the new suggested level of leaf n is lower than the previous one, the processor only removes the votes of leaf n in the balancers from the old suggested level to the new one on the path the processor have traversed from the root to leaf n (lines 1-3 in $Elect2Shrink()$). Otherwise, the processor votes and checks the shrinkage condition for all balancers from the new level to the old one on its path (lines 5-14). Each balancer has a array $votes$ whose size is the maximal number of possible leaves in its sub-tree and each element in the array is reserved for each of its leaves. If the leaf n votes for a balancer b, $b.votes[n]$ is changed from 0 to the current level of leaf n. The value of $votes[n]$ is used to check the shrinkage condition: the processor computes the weight of each of leaf n's votes by the number of leaves at the lowest level the vote represents (line 12). The weight of the sub-tree of a voted balancer is computed by the number of leaves at the lowest level as if the sub-tree were expanded to the lowest level (line 7). The lowest level is $\lceil \log_2(the_maximal_number_of_processors) \rceil$. If the sum of weights of votes for the balancer is more than half

[4]The right side is shown in Figure 1

```
bool Elect2Shrink(node_t *n, node_t *path[]) {
 1  if( n->osLevel < n->sLevel)
 2    for( i = n->osLevel+1; i <= n->sLevel; i++)
 3      path[i]->votes[n->id] = 0;                    /* remove n's votes */
 4  else
 5    for( i = n->sLevel+1; i <= n->osLevel; i++)
 6      path[i]->votes[n->id] = n->level;
 7      bWeight = pow( 2, MaxLevel - path[i]->level); /* the subtree weight */
 8      lWeight = 0;
 9      for all path[i]'s leaves      /* compute the weight of all votes in path[i] */
10        lVote = path[i]->votes[leafId];
11        if( lVote != 0)
12          lWeight += pow( 2, MaxLevel - lVote);
13      if( lWeight / bWeight > 0.5)
14        return( Shrink( path[i], n));
15  return SUCCEED;
}

bool Shrink( node_t *n) {                       /* n: the suggested balancer */
 1  if( !TATAS( &n->rLock, 1))                     /* acquire reaction-lock */
 2    return SUCCEED;
 3  partner = &Counters[n->id];
 4  for( i=0; i<NumPros; i++)          /* check pending processors at partner */
 5    if( Read( &Tracing[i]) == partner)
 6      return FAIL;
 7  if(!acquire_lock_c(&partner->nLock, n->id) || (n->state != BALANCER))
 8    return FAIL;
 9  Reset partner to OLDLEAF state;  /* avoid reactive adjustments at partner*/
10  Lock active leaves and old leaves in the sub-tree
11  of n in increasing order of node Id;
12  pr = n->parent;
13  branch = n->nodeId % 2;
14  if( branch == 0)
15    pr->right = partner;
16  else
17    pr->left = partner;
18  np = 0;
19  for( i=0; i<NumPros; i++)     /* count pending processors in n's subtree */
20    trace_i = Read( &Tracing[i]);
21    if( ((trace_i == n) || IsParent(n->id,trace_i->id))
22        && (trace_i->state in {LEAF,BALANCER}))
23      np++;
24  Set all balancers in the sub-tree to OLDBALANCER;
25  Get the lists of the active leaves and their values,
26  set their state to OLDLEAF and release all leaves;
27  partner->count = CalcCount( n, np, active_l, value_l);
28  partner->state = LEAF;
29  release_lock(&partner->nLock);
30  release(n->rlock);
31  return SUCCEED;
}
```

Figure 9. Elect2Shrink, and Shrink procedures

the weight of its sub-tree, the processor calls $Shrink()$ to shrink the balancer's sub-tree to its corresponding leaf (line 14).

In procedure $Shrink(n, s)$, first the processor called p_i tries to acquire the reaction lock $rLock$ (line 1). If p_i cannot get the lock, it returns $Succeed$ because there is another processor that is trying to shrink the sub-tree of balancer

n to its corresponding leaf. If it acquires the reaction lock of balancer n successfully, p_i must ensure that in the leaf corresponding to balancer n there is no pending processor (lines 4-6). Then processor p_i tries to acquire the node lock of the leaf (line 7).

After successfully acquiring the node locks of the corresponding leaf and all leaves in the sub-tree in increasing order of node identity while the state of n is still $Balancer$ (lines 7-11 in $Shrink()$), processor p_i switches the pointer from balancer n to its corresponding leaf by updating $right_{parent}$ (or $left_{parent}$) to $partner$ (lines 14-17). Then it goes through the sub-tree to count the number of effective processors, i.e. the processors who are there and who can affect the counter value for the new leaf. It must also get the list of $active$ leaves, which are the leaves needed in order to get their current counter values to calculate the next counter value for the new leaf (lines 18-25). The counter value for the new leaf is calculated based on the list of active leaves, their values and the number of effective processors on the balancer's sub-tree (line 27). Procedure $CalcCount()$ is used to calculate the new counter value for the new leaf (line 27). Procedure $CalcCount()$ is implemented as in Figure 10.

```
int CalcCount( int np, list_t L, list_t V) {
  Convert leaves in L to the same level, the lowest
  => new lists of leaves L' and their values V';
  Distribute the number of processors np on the
  leaves in L' so that step-property is satisfied.
  Call the leaf visited by the last processor lastL;
  result = lastL->count - pow(2, lastL->level)
           + pow(2, n->level);
}
```

Figure 10. Calculating the counter value for the new leaf

For example, in Figure 1 if the sub-tree of balancer 2 shrinks to leaf 2 and the list of active leaves is $\{4, 10, 11\}$, leaf 4 needs to be converted to two leaves 8 and 9 at the same level with leaves 10 and 11, the lowest level. Thus, the new list of leaves L' is $8, 9, 10, 11$. After converting, the sub-tree becomes balanced and the step-property must be satisfied on the sub-tree. The following feature of a tree satisfying step-property was exploited to calculate the new counter value in our implementation. Call the highest counter value of leaves at the lowest level $MaxValue$. The counter values of the leaves must be in range $(MaxValue - 2^{LowestLevel}, MaxValue]$.

After assigning the counter value for the new leaf, the processor releases the new leaf (line 29) and the reaction lock of the balancer (line 30).

4 Correctness Proof

In this section, we clarify some subtle points in our algorithm. As mentioned in subsection 3.4 and subsection 3.5, the leaves are locked in decreasing order of leaf identities in $Grow()$ but in increasing order in $Shrink()$ and thus we need to prove that deadlock never occurs. The reactive trees so far [1] have an implicit assumption that processors passing through the tree never fail because lock-based synchronization is used to protect critical sections and so does our self-adjusting tree.

Lemma 4.1. *Self-adjusting trees are deadlock-free.*

Proof. The interference between two balancers who are trying to lock leaves (recall that processors lock leaves on behalf of balancers) occurs only if one of the balancers is the other's ancestor in the *family* tree. Let us consider 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 shrinkage tasks that shrink their sub-trees to leaves, both will lock leaves in increasing order of leaf identities by using the procedure $Acquire_lock_c()$ that conditionally acquires locks. If the leaf with smallest identity that b_j needs is locked by b_i, procedure $Acquire_lock_c()$ 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 itself, b_i must wait at the leaf until b_j completes its own work and then b_i continues locking all necessary leaves. If no processor locking the leaves on behalf of a balancer fails, no deadlock will occur.

Case 2: b_i executes a shrinkage task, which shrinks its sub-tree to a leaf, and b_j executes an expansion task, which expands its leaf to a sub-tree. 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 locked leaf k successfully and is now trying to lock leaf $(k+1)$ whereas b_j locked leaf $(k+1)$ successfully and is trying to lock leaf k. Because b_i and b_j use the procedure $Acquire_lock_c()$ that conditionally acquires the locks, b_j will fail to lock leaf k, which is locked by its ancestor, and will release all the leaves it locked so far (lines 7-11 in $Grow()$). Therefore, b_i can lock leaf $k+1$ and continues locking other necessary leaves. That is, deadlock does not occur in this case either.

Recall that there is no the case that b_i executes an expansion task. This is because in its sub-tree there is at least one pending processor that helps b_j do its reaction task.

□

Corollary 4.1. *In the shrinkage process, if the corresponding balancer successfully acquired the necessary leaf with smallest identity, it will then successfully lock all the leaves it needs.*

Therefore, procedure $Shrink()$ returns $Fail$ only if the leaf corresponding to the balancer is locked by an ancestor of that balancer.

Lemma 4.2. *There is no interference between any two expansion tasks in our self-adjusting tree.*

Proof. (Sketch) Similarly as in 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 *family* 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 the ancestor of the other. □

Lemma 4.2 is reason why we need not to lock balancers before resetting their variables in $Grow()$ (line 6 in $Grow()$).

Secondly, we need to show that the number of processors used to calculate the counter value for the new leaves in both $Grow()$ and $Shrink()$ is counted accurately by using the global array $Tracing$.

Lemma 4.3. *The number of effective processors*[5] *that are pending in a locked sub-tree or a locked leaf is counted accurately.*

Proof. A processor p_i executing an adjustment task switches a pointer from a branch of the tree to a new branch before counting the pending processors in the old branch (lines 15-22 in $Grow()$ and lines 14-23 in $Shrink()$). Because of this and because procedure $Assign()$ is atomic to procedure $Read()$ (cf. lemma 3.1), the processor p_i will count 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.

i) In the case of tree expansion, the number of pending processors in the locked leaf is the number of effective processors.

ii) In the case of tree shrinkage, we lock both old and active leaves of the locked sub-tree so that no pending processor in the sub-tree 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 sub-tree to $OldBalancer$ and $OldLeaf$ respectively before releasing the sub-tree (lines 24,26 in $Shrink()$, and ii) after locking all necessary balancers and leaves, processors continue processing the corresponding shrinkage/expansion tasks only if the switching balancers/leaves are still in an active state, i.e. $Balancer$ or $Leaf$ (line 9 in Grow() and line 7 in Shrink()). Therefore, a pending processor in the locked sub-tree that visited an $OldBalancer$ or $OldLeaf$ will never visit an active $Balancer$ or $Leaf$ in this locked sub-tree. Similarly, a pending processor in the locked sub-tree that visited an active $Balancer$ or $Leaf$ will never visit an $OldBalancer$ or $OldLeaf$ in this locked sub-tree. Hence, by checking the state of the node that a pending processor p_j in the locked sub-tree is visiting, we can know whether the processor p_j is effective (line 22 in $Shrink()$). In conclusion, the number of effective processors pending in a locked sub-tree is counted accurately by procedure $Shrink()$ also in the case of tree shrinkage. □

[5] *Effective* processors are processors that are not in old balancers nor in old leaves of a locked sub-tree. Only the $effective$ processors affect the next new counter values calculated for the new leaves.

5 Evaluation

In this section, we compare our self-adjusting tree with the reactive diffracting tree [1].

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 [1]. However, subsection *Load Surge Benchmark* in [1] described that the reactive diffracting tree sized to a depth 3 tree when they ran index-distribution benchmark [5] 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 cc-NUMA 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 [5] but also gave the best performance for the diffracting trees in our system. Regarding the prism size, 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 [1, 6].

The system used for our experiments was a ccNUMA SGI Origin2000 with sixty four 195MHz MIPS R10000 CPUs with 4MB L2 cache rated at 11.4 SPECint95 and 19.1 SPECfp95 each. The system ran IRIX 6.5. We ran the reactive diffracting tree (RD-tree) and our self-adjusting tree (SA-tree) in the full contention benchmark, in which each thread continuously executed only the function to traverse the respective tree, and in the index distribution benchmark with $work = 500\mu s$ [1][5]. Each experiment ran for one minute and we counted the average number of operations for one second.

5.1 Results

The results are shown in Figure 11 and Figure 12. The right charts in both the figures show the average depth of our SA-tree compared to the RD-tree. The left charts show the proportion of our SA-tree throughput to that of the RD-tree.

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

Regarding the throughput, our SA-tree performs considerably better than the RD-tree. This is because of four algorithmic differences between the SA-tree and the RD-tree:

1) The SA-tree reacts to the contention variation faster with lower overhead as described in Section 2, whereas in the RD-tree the leaves shrink or grow only one level in one reaction step and then the leaves have to wait for a given number of processors passing through themselves before shrinking or growing again.

2) In the RD-tree, whenever a leaf shrinks or grows, all processors visiting the leaf are blocked completely until the reaction task completes. Moreover, some processors may be forced to go back to higher nodes many times before exiting the RD-tree. In the SA-tree, this is avoided with the introduction of the matching leaves.

3) The execution time of the SA-tree is not penalized from the waiting time that comes from the use of the $prism$ construct. Algorithmically, our SA-tree uses the FETCH_AND_OP atomic primitives to keep the contention on the toggle-bits low and consequently do not need to use the $prism$. These FETCH_AND_OP primitives require only two non-overlapped messages to and from the memory module storing the synchronization flag and do not create further network traffic because they bypass completely the cache-coherence protocol [4].

4) The RD-tree depends on clock readings for their reactive decisions. Clock readings are expensive and do not scale. The reactive mechanism of our SA-tree is based on an efficient on-line algorithmic scheme that does not require any expensive system call.

The results of both benchmarks confirm our arguments. In the full contention benchmark, our SA-tree not only performs faster than the RD-tree in all cases from 4 processors up to 32 processors but also scales up better to the number of processors as shown in Figure 11. From the left chart, we can see that our SA-tree throughput is from four times to twelve times greater than that of the RD-tree when the number of processors goes from four to thirty two, whereas the average depth of both trees are nearly the same as shown in the right chart. In this case, the contentions on the leaves of both trees were nearly the same. The higher throughput the SA-tree gained is because it enables processors to traverse the tree faster than the RD-tree.

In the index distribution benchmark with $work = 500\mu s$, which provides a lower-load environment, our SA-tree throughput is from one and half times to 42 times greater than that of the RD-tree as shown in Figure 12. The charts of the average depths of both trees have approximately the same shapes, but the SA-tree expands from half to one depth unit more than RD-tree. This is because the throughput of the former was much larger than the latter, the contentions on the SA-tree leaves were much more than those on RD-tree leaves. This made the SA-tree expand more.

6 Conclusion

The self-adjusting trees presented in this work distribute the set of processors that are accessing them, to many smaller groups accessing disjoint critical sections in a global coordinated manner. They collect information about the contention at the leaves (critical sections) and then they adjust themselves to attain optimal performance. The self-adjusting trees extend the very 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 that have to be set manually and depend on experimentation.
- The reactive adjustment policy does not use any expensive system call, but it is based on an efficient 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.

Therefore, the self-adjusting trees can react quickly to changes of the contention levels, and at the same time offer a good latency to the processes traversing them. Our self-adjusting trees perform faster than the reactive diffracting trees on the SGI Origin2000, a well-known commercial ccNUMA multiprocessor.

References

[1] G. Della-Libera and N. Shavit. Reactive Diffracting Trees. *Journal of Parallel and Distributed Computing* 60(7): 853-890 (2000)

[2] R. El-Yaniv, A. Fiat, R. M. Karp, G. Turpin. Optimal Search and One-Way Trading Online Algorithms *Algorithmica* 30:101-139 (2001).

[3] J. M. Mellor-Crummey and M. L. Scott Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991

[4] D. S. Nikolopoulos and T. S. Papatheodorou. The Architectural and Operating System Implications on the Performance of Synchronization on ccNUMA Multiprocessors. *International Journal of Parallel Programming* Volume 29, No. 3, June 2001

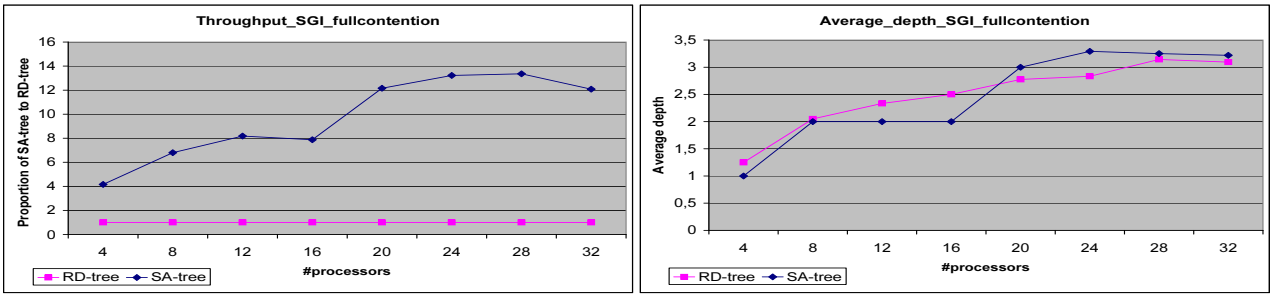

Figure 11. Throughputs and average depths of trees in the full contention benchmark on SGI Origin2000

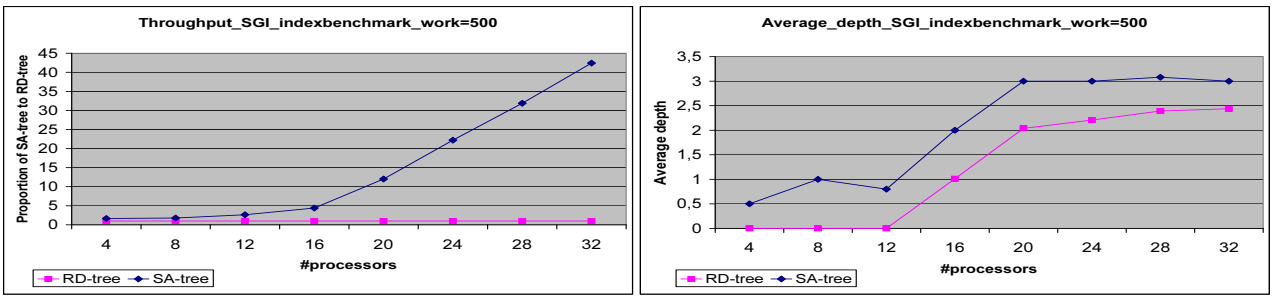

Figure 12. Throughputs and average depths of trees in the index distribution benchmark with $work = 500\mu s$ on SGI Origin2000

[5] N. Shavit, A. Zemach. Diffracting Trees. *TOCS* 14(4): 385-428 (1996)

[6] N. Shavit, E. Upfal and A. Zemach. A Steady State Analysis of Diffracting Trees. *Theory of Computing Systems* 31(4): 403-423 (1998)