Reactive Diffracting Trees

Giovanni Della-Libera
M.I.T.
and
Nir Shavit
Tel-Aviv University and M.I.T.

A shared counter is a concurrent object that provides a *fetch-and-increment* operation in a distributed system. Recently, *diffracting trees* have been introduced as an efficient way of implementing shared counters in heavily loaded systems. Diffracting trees dynamically distribute processors into small groups that can access a collection of disjoint local counters quickly in a globally coordinated way. Their empirical performance under heavy load surpasses all other shared counter implementations. However, diffracting trees of differing depths are optimal for only a limited load range. There would thus be great benefit in designing a diffracting tree algorithm that would effectively scale from a simple cetralized queue-lock based counter at low loads to the optimal size diffracting tree counter as the load increases/decreases.

This paper presents the *reactive diffracting tree* data structure and its implementation on a shared memory multiprocessor system. The reactive diffracting tree is a shared structure similar to a diffracting tree, but which can grow and shrink to better handle changing access patterns and the memory layouts, providing true scalability and locality. The tree mimics the behavior of an optimal size diffracting tree for each concurrency range.

Empirical evidence, collected on the Alewife cache-coherent multiprocessor and the Proteus simulator, shows that the reactive diffracting tree provides throughput within a constant factor of optimal diffracting trees at all load levels. It also shows it to be an effective competitor with randomized load balancing algorithms in several producer/consumer applications.

We believe that reactive diffracting trees will provide fast and truly scalable implementations of many primitives on multiprocessor systems, including shared counters, $k-$exclusion barriers, pools, stacks, and priority queues.

1. INTRODUCTION

Coordination problems in multiprocessor systems have received much attention recently. In particular, shared counters and their applications are an important area of study because the *fetch-and-increment* operation is a primitive that is being widely used in concurrent algorithm design. Since good hardware support for implementing scalable shared counters in multiprocessor systems are not readily available, there have been a variety of software solutions proposed for this problem.

A preliminary version of this paper appeared in the *Proceedings of the 9th Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1997.

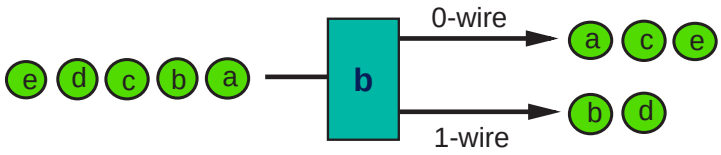

Fig. 1. A Balancer at work

1.1 Background

Straightforward software solutions often involve protecting a centralized counter register with a spin-lock, either a test-and-set lock with exponential backoff (see the work of Agarwal and Cherian, Anderson, Rudolph et. al, and Graunke and Thakkar [2; 3; 27; 13]) or a queue-lock (see the work of Anderson or Mellor-Crummey and Scott[3; 25]). These algorithms are popular because they provide minimal latencies in low load situations, when requests are sparse and mostly sequential in nature. However, they can not hope to obtain good throughput under high loads due to the bottleneck inherent in mutual exclusion.

More sophisticated concurrent algorithms proposed include the combining trees of Yew, Tzeng, and Lawrie [34] and Goodman, Vernon, and Woest [11], the Combining Funnels of Shavit and Zemach [33], the Counting Networks of Aspnes, Herlihy, and Shavit [4], and the Diffracting Trees of Shavit and Zemach [32; 31]. These methods are highly distributed and lower the contention on individual memory locations, allowing for better performance at high loads. The software combining methods [11; 33; 34] have the advantage of providing linearizable [17] counter implementations[1], and under high loads their performance is comparable and even slightly better than that of counting networks [14; 32; 33]. However, empirical studies show that both the software combining methods and counting networks are inferior to diffracting trees under high loads [32]. In such situations diffracting trees are two to three times faster (their average latency for an increment operation is two to three times smaller and throughput is two to three times larger).

What are Diffracting Trees? On an abstract level, diffracting trees are distributed data structures constructed from simple one-input two-output elements called *balancers*. Tokens (processes) arrive on the balancer's input wire at arbitrary times, and are output on its output wires. One may think of a balancer as a toggle mechanism (a bit that is repeatedly complemented), that given a stream of input tokens, repeatedly sends one token to the left output wire and one to the right, balancing the number of tokens that have been output. Figure 1 shows how a balancer would balance five distinct tokens (A-E) arriving sequentially in alphabetical order. A diffracting tree consists of a collection of balancers that are connected to one another by wires to form a balanced binary tree. It distributes input tokens to output wires by having them toggle through a sequence of balancers from the root to leaves. As depicted in Figure 2, tokens are output in increasing order modulo 8 to create

[1] Linearizable counting network constructions exist [15; 22; 24], but require limiting assumptions on the underlying system's behaviour.

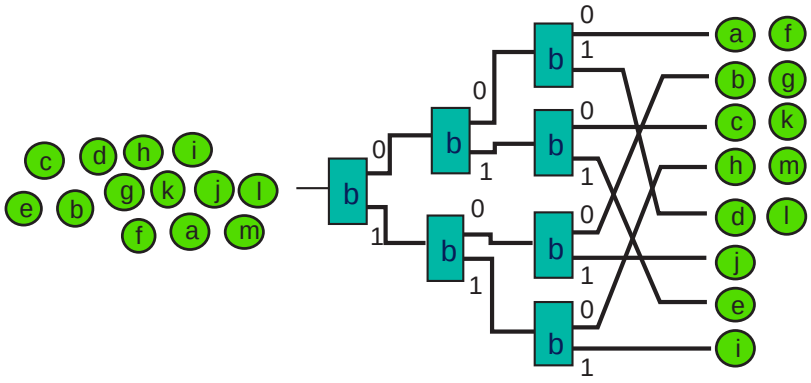

Fig. 2. A Diffracing Tree

a step-like output pattern. Trees of balancers having this step-property can easily be adapted to count the total number of tokens that have entered the network. Counting is done by adding a "local" queue-lock protected counter to each output wire i, so that tokens coming out of that wire are consecutively assigned numbers $i, i+8, i+(8\cdot 2)\ldots$, as in Figure 3.

However, under high loads, the balancer toggle bits, especially the one at the root balancer of the tree, will be accessed by many processes concurrently, forming contention hot-spots [26] and sequential bottlenecks that are as bad as that of a centralized spin-lock-protected counter implementation. Diffracting trees overcome the problem by having a "prism" mechanism in front of the toggle bit of every balancer, allowing independent pairs of tokens to be "diffracted" in separate memory locations in a coordinated manner one to the left and one to the right. A coordinated pair of processors can leave the balancer without either of them having to toggle the shared bit, since each pair of toggles leaves the bit in the same state. The processors need simply agree between themselves which one would have gotten the "0" bit, and which the "1". The diffraction mechanism uses randomization to ensure high collision/diffraction rates on the prism, and the tree structure guarantees correctness of the output values.

Diffracting trees of varying depths provide optimal performance throughout the load range, and the trivial diffracting tree, a counter protected by a queue-lock, provides the best performance under low load. However, a diffracting tree of a given depth has unwanted costs for lower loads due to its higher latencies, and as load increases its performance will eventually levels out as it becomes overcapacitated [31].

In summary, the prior art seems to be firmly divided into two camps: the centralized lock-based algorithms which work well in the low load cases and the distributed algorithms that provide superior performance under high load. The lock-based algorithms are championed by the queue-lock methods, and the distributed algorithms are led by Diffracting Trees. One set of experiments we conducted revealed that in a low load situation, the throughput over a fixed period of time for a queue-lock

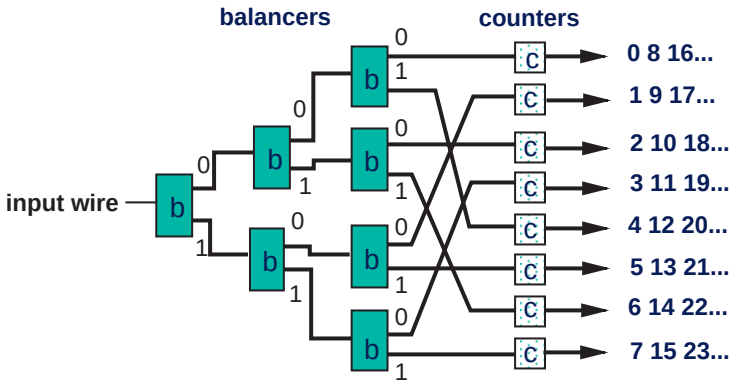

Fig. 3. A counting diffracting tree

based counter was 652 operations while the diffracting tree delivered only 46 operations. With the same period of time but under a high load, the queue-lock counter went down to 595 operations, while the diffracting tree rose to 5010 operations. A factor of 10 difference separates each of these sets of numbers.

1.2 Goals

The family of Diffracting Trees, from a degenerate tree consisting of a single queue-lock based counter, to deep trees with a collection of counters at their leaves, provide peak performance over the full range of concurrency levels. Our goal is to create a single dynamically changing Diffracting Tree structure that can estimate the system's load at any point and assume the optimal size Diffracting Tree for that load.

Lim and Agarwal [20; 21] recently came up with a reactive scheme that switched between a test-and-test-and-set lock [27], a queue lock, and a combining tree. This algorithm performed well from the low to mid-load ranges, as the combining tree took over for the queue lock. The algorithm only applies to algorithms that have one centralized lock-based counter, which precludes diffracting trees, but gave us valuable insights to reactive policy making.

We now focus on the two main ideas necessary to create a reactive diffracting tree algorithm.

—Localize decision making.
—Use cache-coherence to make global agreement inexpensive.

Localized decision making spares processors from continually deciding on the overall structure of the shared counter, which is what Lim and Agarwal's algorithm requires. A major drawback with global decision making is that processors can get delayed while they wait for a change to occur. By making the changes in the shared counter local to only part of the counter data structure, the number of processors directly delayed will drop significantly, and when other processors arrive in the

changed part of the structure, the decision will have already been made and they can quickly adapt.

Cache-coherence makes localized decision making a reality. Keeping processors in agreement globally is usually an expensive requirement. If an algorithm adds global state which does not often change in high load situations, then this information can be cached, making constant reference to it an inexpensive proposition. If the load in an area of the tree is low, then changes can be made without high cost, which enables localized decision making.

1.3 Reactive Diffracting Trees

Our algorithm, the Reactive Diffracting Tree (or RDT), uses the above two principles to make diffracting trees reactive. A significant change in the load of the system will cause the RDT to grow or shrink into the matching optimal tree. These changes occur in a localized fashion at some leaves of the tree, but in a way that quickly applies to all leaves if a genuine global change of load occurred. However, in some cases the tree's memory layout is designed so that different ends of the trees are layed out in separate parts of memory, the tree may become irregularly formed to give optimal performance. State is added to the nodes of the tree to indicate if they are acting as balancers or counters, and the caching of this state information enables processors to pass through the tree without much delay.

We implemented and tested the RDT on the MIT Alewife machine of Agarwal et. al [1]. However, the largest Alewife machine only has 32 nodes, limiting the load range we could test with. We thus show that the Proteus Parallel Hardware Simulator of Brewer et. al [6; 7], which we run with up to 256 processes, simulates Alewife well, giving results that are comparable when normalized. For the experiment described earlier where we measured throughput over a fixed period of time, the single RDT structure provided under low load 243 operations and under high load 3932 operations, as compared with 46 for the optimal queue-lock and 5010 for the optimal diffracting tree. The results we present in the experimental section show that in general the RDT performs within a constant factor of optimal diffracting trees at all load levels, and future work promises to lower this factor.

In summary, we believe that the RDT and its underlying concepts will prove an effective paradigm for the design of future data structures and algorithms for multi-scale computing. This remainder of this paper is organized as follows: Section 2 explains the design of the RDT and its asynchronous shared-memory implementation, and discusses different scaling policies. Section 3 provides the empirical performance results on the Alewife machine and the Proteus simulator. Section 4 lists directions for further research.

2. REACTIVE DIFFRACTING TREES

In order to keep the presentation as self contained as possible, we begin by reviewing the basics of diffracting trees. We then describe in detail the changes necessary to make them reactive. This includes an implementation of the RDT on an asyn-

```
type balancer is
  lock:   lock
  toggle: int /* 0 or 1 */
  Left:   ptr to balancer /* wire y0 */
  Right:  ptr to balancer /* wire y1 */
end

globals
  Width: int
  Root : ptr to root of Binary[width] tree

function simple_balancer(b: ptr to balancer) returns ptr to balancer
begin
  lock(b->lock)
  k := b->toggle
  b->toggle := 1-k  /* toggle the bit */
  unlock(b->lock)
  if k = 0 return b->Left
  else return b->Right
end

function fetch&incr() returns int
begin
  b:= Root
  while not leaf(b)
    b := simple_balancer(b)
  endwhile
  i := increment_counter_at_leaf(b)
  return i * Width + number_of_leaf(b)
end
```

Fig. 4. A Shared-Memory tree-based counter implementation

chronous, cache-coherent, distributed shared-memory system. Finally, the reactive scaling policy issue is discussed.

2.1 Diffracting Trees

The Diffracting Trees of Shavit and Zemach [32] are counting trees, a special form of the Counting Network data structures introduced by Aspnes et. al [4] (See also [8; 18; 19]).

A Counting Tree is a binary tree of nodes called balancers. A *balancer* is a computing element with one input wire and two output wires. Tokens arrive on the balancer's input wire at arbitrary times, and are output on its output wires. We denote by x the number of input tokens ever received on the balancer's input wire, and by y_i, $i \in \{0, 1\}$ the number of tokens ever output on its ith output wire. Given any finite number of input tokens x, it is guaranteed that within a finite amount of time, the balancer will reach a *quiescent* state, that is, one in which the

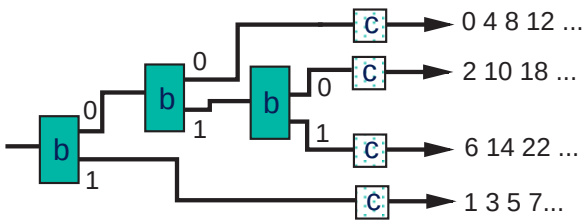

Fig. 5. An irregular diffracting tree's counting scheme

sets of input and output tokens are the same. In any quiescent state, $y_0 = \lceil x/2 \rceil$ and $y_1 = \lfloor x/2 \rfloor$. We will abuse this notation and use y_i both as the name of the ith output wire and as the count of the number of tokens output on that wire. As explained earlier, Figure 1 shows how a balancer would balance five distinct tokens (A-E) arriving sequentially and in alphabetical order.

The Counting Tree layout is defined as follows. Let k be a power of two, and let us define the counting tree BINARY$[2k]$ inductively. When k is equal to 1, the BINARY$[2k]$ network consists of a single balancer with output wires y_0 and y_1. For $k > 1$, we construct the BINARY$[2k]$ tree from two BINARY$[k]$ trees and one additional balancer. We make the input wire x of the single balancer the root of the tree and connect each of its output wires to the input wire of a tree of width k. We then re-designate output wires $y_0, y_1, \ldots, y_{k-1}$ of the tree extending from the 0 output wire as the even output wires $y_0, y_2, \ldots, y_{2k-2}$ of BINARY$[2k]$ and the wires $y_0, y_1, \ldots, y_{k-1}$ of the tree extending from the balancer's 1 output wire as the odd output wires $y_1, y_3, \ldots, y_{2k-1}$.

One can extend the notion of quiescence to trees in the natural way, and define a *counting tree* of width w as a tree of balancers, BINARY$[w]$, with outputs $y_0, .., y_{w-1}$ that satisfy the following *step property*:

In any quiescent state, $0 \leq y_i - y_j \leq 1$ for any $i < j$.

Figure 2 shows a BINARY[8] tree moving input tokens to output wires in increasing order modulo 8 while preserving the step property. The tree counts the total number of tokens that have entered the network by way of the local counters attached to each of the output wires (see Figure 3). Tokens coming out of that wire i are consecutively assigned numbers $i, i + w, \ldots, i + (y_i - 1)w$.

On a shared memory multiprocessor, one implements a balancing tree as a shared data structure, where balancers are records, and wires are pointers from one record to another. Each of the machine's asynchronous processors run a program that repeatedly traverses the data structure from the root input pointer to some output pointer, each time shepherding a new token through the network. Pseudo-code for this program appears in Figure 4. We use an MCS-queue-lock [25] to avoid race conditions on the balancer's toggle bit and on the shared local counters at the leaves.

Diffracting Trees are counting trees whose balancers are of a novel type called

Diffracting balancers. One could easily implement a balancer using a single `toggle` bit. Each processor would toggle the bit inside the balancer, and accordingly decide

```
type balancer is
    spin:   int
    size:   int
    prism:  array[1..size] of int
    toggle: int    /* 0 or 1 */
    lock:   Lock
    Left:   ptr to balancer  /* wire y0 */
    Right:  ptr to balancer /* wire y1 */
endtype

location: global array[1..NUMPROCS] of ptr to balancer

function diff-bal(b: ptr to balancer) returns ptr to balancer
begin
    location[MYID] = b
    forever
        rand_place = random(b->size)
        his_id = Swap(b->prism[rand_place], MYID)
        if CompareSwap(location[MYID], b, EMPTY) then
          if CompareSwap(location[his_id], b, EMPTY) then
            return b->Left
          else location[MYID] = b
        else return b->Right

        repeat b->spin times
            if location[MYID] != b then
                if b->spin < MAXSPIN then
                  b->spin = b->spin * 2
                return b->Right
        end repeat

        if TestTestSet(b->lock) then
          if CompareSwap(location[MYID], b, EMPTY) then
              k = b->toggle
              b->toggle = 1 - k
              release_lock(b->lock)
              if b->spin > MAXSPIN then
                b->spin = b->spin / 2
              if k = 0 return b->Left
              else return b->Right
          else
              release_lock(b->lock)
              return b->Right
    endfor
end
```

Fig. 6. Code for a Diffracting Balancer

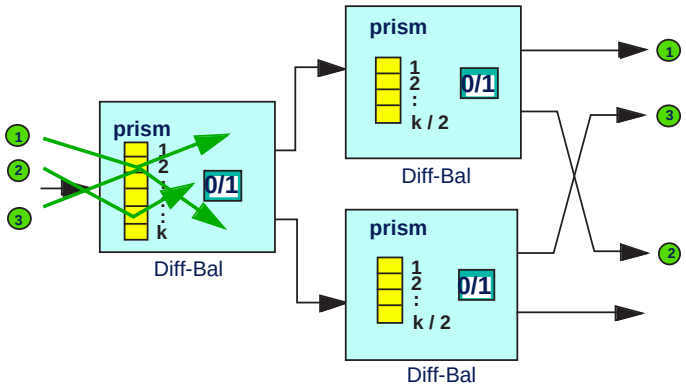

Fig. 7. The diffracting tree mechanisms

on which wire to exit. However, if many tokens attempted to pass through the same balancer concurrently, the toggle bit would quickly become a hot-spot. Even if one applied contention reduction techniques such as exponential back-off, the toggle bit would still form a sequential bottleneck. One can overcome this sequential bottleneck based on the following observation:

> *If an even number of tokens passes through a balancer, they are evenly balanced left and right, yet the value of the toggle bit is unchanged.*

Thus, one can allow pairs of colliding tokens to "pair-off" and coordinate among themselves which is diffracted "right" and which diffracted "left". Then they could both leave the balancer without either of them ever having to touch the toggle bit. By performing the collision/coordination decisions in separate locations instead of a global toggle bit, one can increase parallelism and lower contention. Tokens that do not collide are simply forwarded to access the toggle bit and are routed accordingly as before. To guarantee good performance one must make sure that many collisions and thus few toggle accesses occur, not an obvious task given the asynchrony in the system.

To achieve this goal, Shavit et. al [32; 31] provide an efficient implementation of the diffracting balancer that is based on adding a special `prism` array "in front" of the toggle bit in every balancer. When a token (processor) P enters the balancer, it first selects a location j in `prism` uniformly at random. P tries to "collide" with the another processor that selected j, and if successful they leave the balancer one to the left and the other to the right. Otherwise, P waits ("spins") for a fixed time `spin` to see whether some other processor R will enter and collide with it by selecting the same location j in `prism`. If no collision occurs within time `spin`, P attempts to access the queue-lock on the toggle, and if it fails, it starts all over again by accessing the prism [31]. Figure 6 Figure 7 shows three tokens traversing the tree, where tokens 1 and 3 collide on the first prism and are routed left and right and token two does not collide and is thus routed to the toggle bit which routes it to the left. Figure 6 shows the code for the diffracting balancer. This algorithm

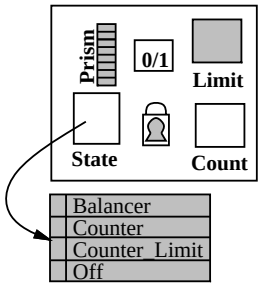

Fig. 8. A reactive diffracting tree node

has been shown to satisfy the balancing properties above [32].

2.2 Reactive Diffracting Trees

The steady-state analysis in [31] showed the importance of relation among the depth of the tree, the size of the prisms, and the spin constant to tree performance. One of the findings of their analysis is that a tree designed to serve P processes should have a depth d and number of prism locations L such that $\frac{P}{dL} = O(1)$, $L \leq d$, and $L = cd2^d$, where c is a machine-dependent constant. Their idea was that given the approximate range of load that a diffracting tree would be subject to, a developer could choose appropriate values of d and subsequently decides on an appropriate L. A reactive backoff scheme was also implemented in [31] to provide the best spin constant.

The idea behind our new reactive algorithm is to move from the fixed `Binary` structure to a flexible tree structure, one that would allow dynamic control of the three parameters: Tree Depth, Prism Size, and Spin. The new structure will craft the optimal diffracting tree for a given load. It will employ localized decision making to allow the tree to change depth. As the number of processors P and subsequently the load changes, the tree will expand or collapse to optimize d and L. The spin constant will be dynamically determined using the reactive backoff scheme described in [31].

We now describe a sequence of changes to the original diffracting tree algorithm in order to turn it into a reactive diffracting tree. A formal I/O automata based specification and correctness proof of this algorithm is outside the scope of this paper and can be found in Della-Libera's thesis [10].

2.3 Irregular Diffracting Trees

We begin by relaxing the restriction that a counting tree is a balanced binary tree. We only require that balancers in the tree have two children and counters be located only at the tree leaves. The idea is that we can avoid the expense of a requiring processors to agree to "globally" switch from one diffracting tree to another of an entirely different size, if we can "locally" shrink and grow the tree from its leaves. However, expanding or shrinking locally will cause the tree to be irregular at times,

```
typedef State oneof Balancer, Counter, Counter_Limit, or Off

type node is
  node_lock:    Lock
                       /* state and versioning section*/
  state:        State
  ID:           int    /* version of node */
  PID:          int    /* version of children */
                       /* balancer section */
  spin:         int
  size:         int
  prism:        array[1..size] of int
  toggle:       int
  toggle_lock:  Lock
                       /*counter section */
  level:        int
  count:        int
  init:         int
  change:       int
                       /* counter_limit section */
  limit:        int
                       /* binary tree section */
  Left:         ptr to node  /* wire y0 */
  Right:        ptr to node  /* wire y1 */
  Parent:       ptr to node
  Sibling:      ptr to node
endtype
```

Fig. 9. Definition of RDT node structure

and so we show how to design an irregular counting tree that counts correctly.

The idea is simple. In a balanced `Binary` counting tree of depth d, each balancer is the root of a subtree of balancers with counters at their leaves. This subtree can abstractly be viewed as an implementation of a single shared counter that hands out the set of indexes corresponding to the values that would be returned by the counters at its leaves. The complete subtree could thus be replaced by a pointer to a single lock-based counter in place of the root balancer, and the tree would behave exactly the same. Figure 5 shows an example of an irregular tree equivalent to `Binary[8]` and how one would set the counter's returned values to make it hand indexes out correctly.

In more detail, one can assign each node a unique name by associating with it a binary string s determined by the path leading to it from the root of the tree as in the code of Figure 6. The string has, reading from left to right, a 0 digit for each traversal to a left child and 1 for a traversal to a right child. Let $level(s)$ be the number of digits in s and $init(s)$ the positive integer whose binary representation is s, with the least significant bit on the left. For example, in Figure 5 the counter with path $s = 010$ handing out numbers $2, 10, 18...$ has *init* 2 and *level* 3 and the

counter with path $s = 1$ handing out numbers $1, 3, 5...$ has *init* 1 and *level* 1.The counter in a tree consisting of a single root counter would have *init* 0 and *level* 0. It is now easy to see that in order to hand out correct sets of values a counter s in an irregular tree must hand out in increasing order values from the set:

$$Values(s) = \{init(s) + i * 2^{level(s)} : i \geq 0\}$$

The code implementing this formula appears in Figure 11.

Though our goal is dynamically changing trees, we note that irregular trees sometimes have an advantage even under static conditions. Locality issues may cause parts of a balanced binary tree to be slower than others because of the memory layout of the data structure. In such cases it pays off to make the tree irregular to maximize performance on that given memory layout, and indeed in such cases one sees a performance improvement using an irregular tree.

2.4 Growing and Shrinking the Trees

We now describe the design of a new algorithm for a given node in the reactive diffracting tree. The key idea is that a node will contain the necessary state information in order to function either as a Balancer or a Counter. We will thus be able to change its functionality from one to the other based on a local load measure. If a node is a counter and its load increases, it will "grow" by unfolding into a subtree consisting of a balancer and two counters as its `Left` and `Right` children. If it is a balancer with two children counters and the load on it decreases, it will "shrink" by folding its two children counters and changing to function as a counter.

The first issue that arises when attempting to implement such a dynamic structure is that it is no longer possible for every processor to initially memorize the complete structure of the tree, as it could in the static diffracting tree. Thus, upon visiting a node of the tree, a processor would need to determine if that node was a counter or balancer. We do so by adding to each node a shared `state` variable. This `state` variable takes on three values, Counter, Balancer, or Off. The first two are clear in the context of merging the types of data structures together. A processor that visits a Balancer node balances, and one that visits a Counter node counts. The Off state is used in order to avoid the need to dynamically allocate/deallocate memory for nodes as the tree grows and shrinks (allocation could take a long time and bottleneck the processors, or a pointer to a deallocated node could remain around long enough to cause a problem if it was reallocated). Instead, one pre-assigns a maximal tree size creating an initial configuration of balancers and counters from the root and leaving the rest of the nodes in the tree Off. To unfold a node it is set to Counter and when it is again folded it is turned back to Off. A description of the node structure is provided in Figure 8 (The additional Counter_Limit state is explained in the sequel).

The above scheme raises a key performance issue. The reason for the good performance in diffracting trees is the highly distributed nature of the data structure [32]. By having processors access disjoint memory locations contention is lowered to a minimum. However, the `state` variable introduces a potential source of con-

```
1   root: global ptr to node   /* main root of tree */
2   Bookkeeping: global array [1..NUMPROCS] of pair
3
4   function fetch_incr() returns int
5       answer:   int
6       IDRecord: array [enumeration of nodes] of int
7       n:        ptr to node
8   begin
9       IDRecord[root] = 0
10      n = root
11      answer = INVALID
12
13      forever
14          if (n->ID != IDRecord[n]) then
15                 n = n->Parent
16                 continue
17
18          switch n->state
19            case Balancer:
20              Bookkeeping[MYID] = <n>
21              if ((n->state != Balancer) || (n->ID != IDRecord[n]))
22                 n = n->Parent
23                 continue
24              IDRecord[n->Left] = n->PID
25              IDRecord[n->Right] = n->PID
26              if n->state == Balancer then
27                n = diff-bal(n)
28            case Off:
29              n = n->Parent;
30            case Counter or Counter_Limit:
31              answer = increment_counter(n);
32              if valid(answer) then
33                return answer
34              else n = n->Parent
35          endswitch
36      endfor
37  end
```

Fig. 10. Code for main traversal of RDT

tention since all processors accessing a given node in the tree will attempt to read it concurrently. The way to overcome this problem and achieve good performance is to use a cache coherence mechanism, either explicitly written or one provided by the multiprocessor machine, in order to keep an updated copy of each node's state information locally. The idea is that if the cache coherence mechanism works correctly, by adding a sensible scaling policy controlling tree size, state variables of nodes in the upper parts of the tree (closer to the root) will most of the time have valid cached copies and so the shared copies will rarely be concurrently accessed.

Unfortunately, with caching in place, there is a time delay until all caches are

```
function increment_counter(n:ptr to node) returns int
    answer:int
begin
    acquire_lock(n->node_lock)
    if (n->state == Counter or Counter_Limit) and
                        (n->ID == IDRecord[n]) then
        answer = n->count
        n->count = n->count + power(2,n->level)
        if n->count == n->limit then
            n->state = Off
            n->ID = n->ID + 1
        release_lock(n->node_lock)
        return answer
    else
        release_lock(n->node_lock)
        return INVALID
end
```

Fig. 11. Code for counting in an RDT node

updated after a `state` change (an invalidation) [3]. This may cause a situation in which a process accesses a node only to discover that it is in the Off state. We overcome this problem by noticing that if a processor visits an Off node, it follows that the tree folded beneath it. The solution is thus for a processor to trace up the node's ancestral path until it finds a non-Off node it can successfully visit, eventually leaving the tree after accessing a some counter. Since each traversing process accesses a counter once, even after backtracking, the remaining issue to be dealt with is how to efficiently maintain consistency of the values returned by the dynamically changing counters. In other words, how does one prevent duplicating or omitting of values.

To better understand how consistency could be violated with the current design, consider the following scenario. Consider an RDT in the initial shape of a `Binary`[2] tree, consisting of a simple balancer at the root and two child counters. Clearly, one counter hands out the even numbers and one the odd numbers. Assume that the first number to be handed out is 0, and there are 10 increment requests made. 5 requests leave the root balancer along each of the two wires, but do not access the counters yet. Next, the tree decides to shrink, turning into a Counter at the root and turning the two children Off. Assume that the 5 requests along one wire arrive at the "even" counter, see the Off state, and return to the parent, obtaining the first 5 values, 0,1,2,3, and 4. Now, the tree decides to unfold again, and it initializes the two Counters to next hand out values 5 and 6. Finally, the other 5 requests access the odd counter. They receive 5, 7, 9, 11, and 13. Notice the problematic gaps between the 10 numbers handed out.

To solve this problem, we need to make sure that once a balancer becomes a counter, all requests that have passed through that balancer and have not been satisfied return back to the same node and access it again. The way to do this is

to use a *versioning scheme*. When a balancer becomes a counter, the two children will increase their *version numbers*, so that if a processor arrives at a node with a different version number[2] from that which it expected given its parents version number, it will revisit the parent node and get updated. To make versioning efficient, each processor caches the versioning numbers throughout its traversal of the tree, and if at any point it finds an inconsistent version number, it traverses back up the tree until it finds agreeing version numbers, which in the worst case will require traversing back to the *Root* node whose version number never changes. To reduce size and complexity in the implementation, the versioning scheme can be folded into the state variable , since versioning really is additional state. However, for simplicity, it is kept separate here. The definition of the new node structure is given in Figure 9.

Figure 10 contains the code for the main traversal of a processor through a reactive diffracting tree. The Bookkeeping work is explained later in this chapter.

2.5 A Walk Through the Traversal Code

The traversal starts with some base initializations, starting the traversal at the root node (Lines 9-11). The traversal is structured as an infinite loop (Lines 13-36) that continues until an answer is found and the loop is broken. The first thing that is checked is whether a processor's view of the version (`IDRecord[n]`) of the current node is equal to the node's actual version (`n->ID`). If it is not, the processor moves up the hierarchy to `n`'s parent and restart the loop (Lines 14-16).

It then enters the main part of the loop which is a switch of the current node's state (`n->state`) (Lines 18-35). If the node is a balancer, the processor performs some bookkeeping to track what node it is at (Line 20), does a sanity check to make sure the checks it performed are still valid (Lines 21-23), then stores away what it expects the node's children's versions to be when it reachs them. (Lines 24-25). Finally, it performs the balancing step and continues the loop with `n` becoming one of the nodes children (Lines 26-27). The next part of the switch is the Off state, which means the processor reached a node after it was folded. In that case it simply moves up to `n`'s parent and continues (Lines 28-29).

Finally, the processor reaches the case where it hits a Counter or Counter_Limit. It attempts to increment the counter (Line 31). If it is successful, then it exits the function with the correct answer (Line 33), otherwise it moves up to `n`'s parent and continues (Line 34).

2.6 The Folding Mechanism

We now describe how the transitions that reshape the tree structure work. The *folding* transition occurs locally at the bottom of the tree: two sibling counters fold into their parent balancer, becoming a new counter. The *unfolding* transition

[2]Technically, these version numbers are integers from an unbounded range, but they are bounded by the values of the counters, so any implementation which handled the overflow of the counters could handle this as well.

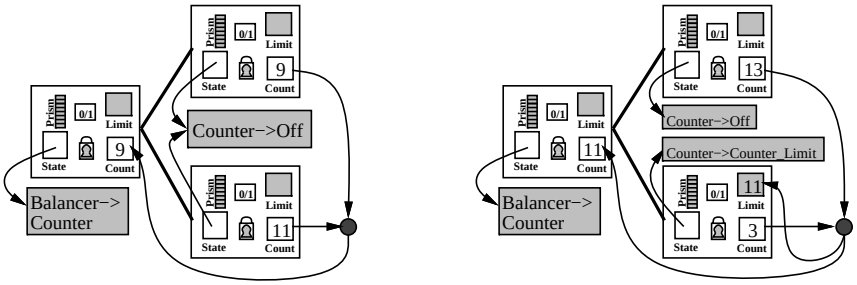

Fig. 12. Two cases of Folding, Parent node at Level 1

transforms a counter at the tree leaf into a balancer with two counters as children. The algorithms we present here lock all of the three nodes involved to perform a transition, but this can be reduced to having one lock at a time. For the sake of simplicity we avoid the reduced version which requires extra state information and significantly complicates the transition code.

We begin by describing the algorithm's folding mechanism, the code for which appears in Figure 14, but leave for later the discussion as to how a process decides to trigger folding. Upon deciding that a balancer and its children counters need to be folded, a processor will attempt to acquire the three node locks. If it is successful, then it tests whether the three nodes are a balancer with two child counters. Once the locks are obtained and the states are checked, the two child counters' values are compared. Now, the values these two counters hand out are alternating values their parent would have handed out as a single counter. Imagine the ordered sequence *Values* of indexes that their parent would hand out if it was a single counter. By definition one of the child counters hands out the values which appear in the odd positions of the sequence, and the other hands out the even-positioned values. The ideal situation in folding is that the two counters' current values are consecutive in the list *Values*. The *init* value, the next value to be handed out by the parent counter, can be set to the lower of the two. The children can then be turned Off, and the parent can thus be set to Counter. This is demonstrated by the right-hand picture in Figure 12, describing such a folding transition with a parent node at *level* 1.

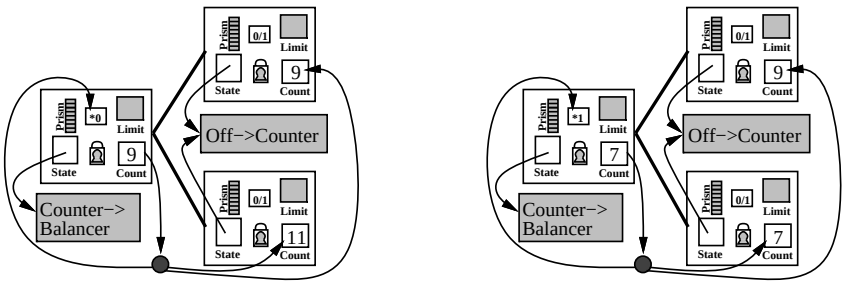

Fig. 13. Two cases of Unfolding, Parent node at level 1

There are however cases where the two counters' values are not consecutive on the sequence of *Values*, and in which additional reasoning is required. For example, consider a situation as in the righthand part of Figure 12 where the parent is at *level* 1 and the larger child-counter (incrementing by 4 each time) handed out the value 13. The smaller counter handed out 3. The larger counter must have thus handed out 5 and 9. However, we cannot set the limit of the parent to 13 without guaranteeing that 7 and 11 that should be handed out by the smaller counter will be handed out. A scheme could be implemented that allowed for storing the values that weren't handed out in a queue in the new counter node, but this would add another level of complexity and significantly decrease performance. We thus need an idea that allows the algorithm to transition quickly without centralized accounting for the unreturned values.

We proceed as follows. After locking the three nodes, we take the maximum of the two child-counter values, find its position in the parent's *Values* sequence, and pick the value in the preceding position as a special *limit* value. When folding, the parent counter is assigned the *limit* as its *init* value. The larger counter is turned Off. The reasoning is that it has clearly handed out the value preceding the *limit* (recall that counters alternate in handing out values in the parents *Values* sequence) and has not handed out its current maximum value. By turning it to Off we are at worst forcing processors accessing it to go back to the parent, but no values past and including the limit will be handed out by mistake.

Finally, the state of the smaller counter is set to a new Counter_Limit state, which acts just like a Counter, except that if the counter's value reaches the *limit* value, it turns Off and hands out no more values (including the *limit* which is not handed out). The reasoning for this is that by the definition of the step property for the parent node balancer in the tree (before the locking mechanism of the folding started), there must be at least enough pending processors to traverse the smaller counter to increase its count to reach the *limit* value (by definition it is the smaller counter that hands out the *limit* value).

Let us return to the right-hand part of Figure 12 which depicts an example of one such situation where the larger child-counter incrementing by 4 each time handed out the value 13. The smaller counter handed out 3. The parents sequence of *Values* end with $\{\ldots, 3, 5, 7, 9, 11, 13\}$. The *limit* is thus set to 11, which would have been handed out by the smaller child counter. We know the larger child counter already returned 5 and 9 before setting its current count to 13. This means that the smaller child counter still has to return 7 to make the output sequence complete with a new *init* value of 11. Fortunately, we know that there must be a processor with a pending request that will access the smaller child-counter to get this 7. This is because the larger child-counter's transitioning from 5 to 9 and 9 to 13 requires four tokens (processors) diffracting through the balancer, that is, at least two tokens sent in the direction of the smaller child-counter.

2.7 A Walk Through the Folding Code

Figure 14 contains the folding code. A processors starts the function by acquiring the three node locks so that it can perform state changes atomically (Lines 8-

```
1   function attempt_fold(n:ptr to node) returns boolean
2       nLeft, nRight, nMax, nMin: ptr to node
3       valLimit: int
4   begin
5       nLeft = n->Left
6       nRight = n->Right
7
8       acquire_lock(nLeft->toggle_lock)
9       acquire_lock(nRight->toggle_lock)
10      acquire_lock(n->toggle_lock)
11
12      if (n->state == Balancer) and
13         (nLeft->state == Counter) and (nRight->state == Counter) and
14         ((nLeft->count != nLeft->change) or (nRight->count != nRight->change)) then
15
16          n->state = Counter
17          n->PID = n->PID + 1
18          valLimit = MAX(nLeft->count,nRight->count) - power(2,n->Level)
19          n->count = valLimit
20          n->change = n->count
21
22          Assign nMin, nMax to be nLeft, nRight,
23            such that nMin->count < nMax->count
24
25          nMax->state = Off
26          nMax->ID = nMax->ID + 1
27
28          if nMin->count < valLimit then
29              nMin->state = Counter_Limit
30              nMin->limit = valLimit
31          else
32              nMin->state = Off
33              nMin->ID = nMin->ID + 1
34
35          release_lock(nRight->node_lock)
36          release_lock(nLeft->node_lock)
37          release_lock(n->node_lock)
38          return TRUE
39      else
40          release_lock(nRight->plock)
41          release_lock(nLeft->plock)
42          release_lock(n->plock)
43          return FALSE
44  end
```

Fig. 14. Code for folding

10). It then checks the conditions necessary for folding: that the parent of the current node is a balancer, both its children are counters, and at least one child has done some counting (Lines 12-14). Then, the parent node is updated with the new state information. It becomes a Counter, it's PID (parent's view of child's ID) is incremented, and it receives a new `value`, which is calculated by taking the maximum of it's two children's values and subtracting one increment from it (Lines 16-20). The processor then sorts out which of the two children has a larger counter value (Lines 22-23). The larger child (`nMax`) is turned Off and its ID is incremented (Lines 25-26). If the smaller child (`nMin`) is in balance with the bigger child, then its turned Off and it's ID is incremented (Lines 32-33). Otherwise, it becomes a Counter_Limit and it's limit is set (Lines 28-31). Finally, whether the folding was done or not, all locks are released and the appropriate value is returned (Lines 33-41).

2.8 The Unfolding Mechanism

Unfolding is a bit simpler than folding, but has its own challenges. Figure 15 contains the code for unfolding. The same three locks are initially set, the node states are checked, making sure that only a node that is currently a Counter with two Off children is unfolded. Then one of the child counters is set to the current counter's value and the other child counter is set to the counters following value. One would want that upon changing the current parent counter to a balancer, the next request always goes to the smaller child counter value. We thus set the balancer's toggle bit in the direction of the child with the smaller value. An example of the two possible cases for a parent node at *level* 1 is shown in Figure 13. [3]

The biggest problem with unfolding is mainly an implementation issue. Consider when the folding and unfolding actions would occur. Folding occurs when there was a below-average load in a given area of the tree. There is thus little delay and contention once the node locks are acquired, since there just aren't that many processors around. On the other hand, unfolding can be a costly process, since it occurs because of high load in a given area of the tree.

To minimize this problem, our implementation releases the parent lock as soon as it's `state` is set, so that the processors that are waiting to access the counter can sooner find out that it is now a diffracting balancer and can be accordingly routed to the child-counters. We also added an optimization that has the processor releasing the parent node's lock go through and tell all of the processors waiting in the queue-lock leading to the node's counter that the state has changed, so they can diffract without delay. This gives the diffracting balancer a good start under high loads. A future optimization could lie in implementing a tree lock [25] instead of a queue lock so this release could occur even faster.

[3] An alternative to the above algorithm would be to keep consistent the role of the Left or first child as the primary child in balancing. Then, the processor returning the smaller value would always be routed there. Since the toggle bit would then always be reset to 0, the change amounts to a technical difference. We chose the first method because it allows each node to have a consistent set of *Values*, simplifying the formal reasoning about how it works [10].

2.9 A Walk Through the Unfolding Code

Figure 15 contains the unfolding code. A processor executing the unfolding function first caches the node's current version and then takes care of some bookkeeping, mainly checking to see whether any processors are currently diffracting through this node. If this is the case the processor gives up (Lines 7-9). Otherwise, it acquires the locks for this node and it's children to guarantee atomicity (Lines 11-13). It checks the conditions needed for unfolding, namely that the node has been used for counting, is a Counter, both children are Off, and the version is consistent with the one cached at the beginning of the unfolding function's execution (Lines 15-17). It then updates the parent node's information, making it a Balancer and incrementing it's PID (parent's view of child's ID) by one (Lines 19-20). Then, based on the final value of the parent node, it decides which child to initially point to (Lines 21-25). Finally, the parent's lock is released so that other nodes can start balancing (Lines 26).

The children's information is then updated. Both become Counters, their ID's are incremented, their values are set based on the final value of the parent node, and their changed values are recorded for future reference (Lines 28-39). Finally, regardless of whether unfolding was completed, all locks are released and the appropriate value is returned (Lines 41-48).

2.10 The Bookkeeping Mechanism

The missing item from the unfolding section was the Bookkeeping mechanism. We now explain its necessity. A Balancer, upon folding into a Counter, must force all of the delinquent processors that were already diffracted through the node to return and traverse it again. Our use of versioning guarantees this will happen. However, imagine that the said node now wishes to unfold again. When it becomes a Balancer, the new processors will balance through it and have correct forecasts for the new child Counters. However, consider a processor currently traversing the node that had also traversed this node in its first incarnation as a Balancer. It received the forecast for its children, then also Counters, and began to balance. While it was attempting to diffract or access the toggle bit, all of these state changes occured, and the node was now into its second incarnation as a Balancer. If this old processor were to diffract against a new processor, then it would upset the balance of the system, since it would arrive at one child Counter, find an incorrect version number, and return to the parent, while its partner from diffraction would arrive at the other child Counter and correctly access it. Now, imagine this happened potentially many times, and each time the old processor went towards the same Counter. When it came time to fold again, there would be no processors left to bring the troubled Counter back into balance with its sibling.

The solution to this problem is simple and the code is given in the main traversal (Figure 10) and unfolding code (Figure 15). We create a global bookkeeping array, one in which every processor has an entry. A processor, upon visiting a Balancer, registers in its entry of the array the balancer it is visiting. It then rechecks to make sure that the node is still a Balancer with the forecasted ID and enters the balancing

```
1   function attempt_unfold(n:ptr to node) returns boolean
2       nLeft, nRight: ptr to node
3       val, ID, i: int
4   begin
5       nLeft = n->Left
6       nRight = n->Right
7       ID = n->ID
8       for i from 1 to NUMPROCS
9           if (Bookkeeping[i] = n) return FALSE;
10
11      acquire_lock(nLeft->toggle_lock)
12      acquire_lock(nRight->toggle_lock)
13      acquire_lock(n->toggle_lock)
14
15      if (n->state == Counter) and (n->count != n->change) and
16         (nLeft->state == Off) and (nRight->state == Off) and
17         (n->ID == ID) then
18
19          n->state = Balancer
20          n->PID = n->PID + 1
21          val = n->count
22          if ((val - n->init) / power(2,n->level)) mod 2 == 1
23              n->toggle = 1
24          else
25              n->toggle = 0
26          release_lock(n->toggle_lock)
27
28          nLeft->state = Counter
29          nRight->state = Counter
30          nLeft->ID = nLeft->ID + 1
31          nRight->ID = nRight->ID + 1
32          if ((val - n->init) / power(2,n->level)) mod 2 == 1
33              nRight->count = val
34              nLeft->count = val + power(2,n->level)
35          else
36              nLeft->count = val
37              nRight->count = val + power(2,n->level)
38          nLeft->change = nLeft->count
39          nRight->change = nRight->count
40
41          release_lock(nRight->toggle_lock)
42          release_lock(nLeft->toggle_lock)
43          return TRUE
44      else
45          release_lock(nRight->toggle_lock)
46          release_lock(nLeft->toggle_lock)
47          release_lock(n->toggle_lock)
48          return FALSE
49  end
```

Fig. 15. Code for unfolding

```
function increment_counter_wrapper(n:ptr to node) returns int
    startTime, elapsedTime, answer, timeRatio : int
    success : boolean;

    begin
        startTime = GetTime()
        answer = increment_counter(n)
        elapsedTime = GetTime() - startTime

        TotalTime[n] += elapsedTime
        TotalHits[n] += 1

        if (TotalHits[n] > MINIMUM_HITS)
            timeRatio = TotalTime[n] / TotalHits[n]
            if (timeRatio > UNFOLDING_LIMIT)
                success = unfold_node(n)
            else if (timeRatio < FOLDING_LIMIT)
                success = fold_node(n)

            if (success)
                TotalTime[n] = 0
                TotalHits[n] = 0

        return answer
    end
```

Fig. 16. Code for Scaling Policy

section of the code. If the information on the second check was inconsistent with the processor's "recollection" of its state in its former traversal of this node, it will go up the tree until it gets back on track. Now, the final piece is a restriction on unfolding. In order for a processor to unfold a Counter, it must traverse the bookkeeping array and make sure no processor is registered as visiting this node as a Balancer. If this traversal is successful, it then unfolds the node if the node's ID was the same as before the bookkeeping array traversal. This scheme is correct for the following reason. Consider the two events: (1) a processor sees that the node is a Balancer and puts itself into the node's bookkeeping array location, and (2) an unfolding processor sees that the node was a Counter and that the first processor's bookkeeping entry did not have this node registered. The ordering of these two events on a machine that guarantees atomicity per memory location, (as the Alewife machine does [9]), implies that either the first processor will see the update upon its recheck, or the unfolding processor will learn that it was out of date and will not unfold.

2.11 Reactive Cache Sizing

The choice of cache size for the algorithm is an interesting study topic on its own. The steady-state analysis [31] predicts that there should be $cd2^d$ prism locations in

the diffracting tree, where c is a constant and d is the depth. Now, in the reactive tree, we have changing depths. The solution is thus for each processor to keep an estimate of the tree's current depth. A skeletal cache of the state is enough for a processor to average the depths of the various paths into the tree and come up with an average depth. Given the best experimental constant c and a large enough prism to handle the largest allowed tree, processors can then simply pick a value randomly within their expected prism size. In practice eventually processors will converge to the same average depth providing the most efficient balancing regardless of the size of the tree.

2.12 The Reactive Folding and Unfolding Policy

There are three main qualifications that a good scaling policy should meet.

—The policy should react quickly to large changes in the load.

—The policy should keep the overhead that it causes low and factor it into its decision making process.

—The policy should keep the number of "false positives," that is, unnecessary changes in tree size, low and limit many consecutive oscillations.

Keeping this in mind, most of our policy development was focused on studying the contention at a counter lock. We felt that this was a good estimate for the overall load of the tree. If the lock was always empty when a processor arrived at the counter, then that counter should be folded into its parent. If the lock was always overloaded, then the counter should be unfolded. We found that observing the time it took to access the counter was a good measure. MCS queue-locks [25] have the nice property that the times measured are stable under consistent load levels, unlike the oscillating times a spin-lock [3] would provide.

We designed our policy by setting thresholds for these times. Passing below a folding threshold or taking longer than the unfolding threshold was a good indication that the local area should change. However, the data structure should not change based on the "opinion" of one processor. Our final policy is thus a variant of Lim's policy in his reactive data structure paper [20]. It uses a string of consecutive times to decide if to allow a transition. The minimum number of consecutive times is a constant that is decided upon by experimentation. The code for implementing this policy appears in Figure 16.

This policy met all three qualifications. A large change in the load will move the time consistently below or above these thresholds and allow for a change. The overhead will be low since only one test is needed to see if the time is within the thresholds, stopping any current streaks and allowing the processor to continue. Finally, by requiring consecutive times, a nice hysteresis affect occurs, because it could not immediately change back in the other direction.

This choice of policy is a simple example that works rather effectively, as confirmed in the experimental section, yet there is room for improvement. Future approaches could include on-line competitive schemes [23] or policies that measure

overall the balancer performance. Since the balancers are tuned by the dynamic prism sizing, a study of the toggling behavior and diffracting rates could reveal a pattern which indicates when it should fold and when its children should unfold.

3. EXPERIMENTAL RESULTS

We evaluated the performance of an implementation of Reactive Diffracting Trees on a shared-memory multiprocessor machine. The MIT *Alewife* machine developed by Agarwal et. al. [1] was our target machine. However, the largest *Alewife* has only 32 nodes, and we were interested in scalability for more than 100 processors. We must thus rely on simulations to provide higher concurrency level results. To support such testing, we conducted the same experiments on the *Proteus*[4] simulator, developed by Brewer et. al. [7], where we were able to extend our results to 256 processes. We performed a correlation study to show that the results were comparable. The set of benchmarks under which we tested RDT include index-distribution, sudden spikes in load levels, and producer/consumer runs, all of which demonstrate the advantages (and disadvantages) of the RDT.

3.1 Experimental Environments

The MIT Alewife machine [1] consists of a multiprocessor with cache-coherent distributed shared memory. Each node consists of a Sparcle processor, an FPU, 64KB of cache memory, a 4MB portion of globally-addressable memory, the Caltech MRC network router, and the Alewife Communications and Memory Management Unit (CMMU). The CMMU implements a cache-coherent globally-shared address space with the LimitLESS cache-coherence protocol [9]. The LimitLESS cache-coherence protocol maintains a small, fixed number of directory pointers in hardware, and relies on software trap handlers to handle cache-coherence actions when the number of read copies of a cache block exceeds the limited number of hardware directory pointers. The current implementation of the Alewife CMMU has 5 hardware directory pointers per cache line.

The synchronization primitives that Alewife provides for performing a read-modify-write operation are based on a set of *colored load/store* operations [1]. Using such operations Alewife supports a memory with *full/empty bits*: every memory location has a full/empty bit associated with it and has operations which allow a processor during a load or store to atomically set the bit to full if empty and vice-versa. This allows mutually exclusive access to the information in the location.

The simulation part of our work was performed using *Proteus*, a multiprocessor simulator developed by Brewer et. al. [7]. We simulated a distributed-shared-memory multiprocessor similar to the MIT *Alewife* machine. Proteus simulates parallel code by multiplexing several parallel threads on a single CPU. Each thread runs on its own virtual CPU with accompanying local memory, cache and communications hardware, keeping track of how much time is spent using each component. In order to facilitate fast simulations, Proteus does not perform complete hardware

[4]Version 3.00, dated February 18, 1993

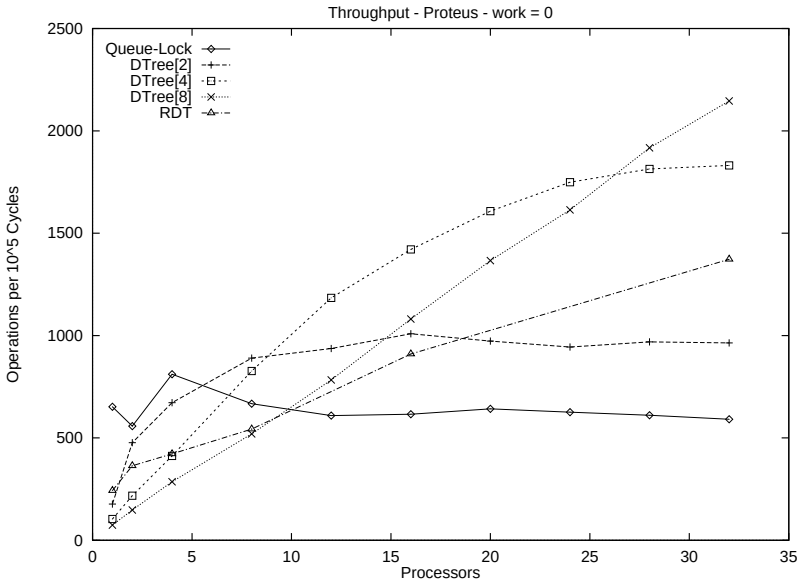

Fig. 17. Diffracting Trees, Queue-Lock Based Counter, and RDT on Proteus

simulations. Instead, operations which are local (do not interact with the parallel environment) are run directly on the simulating machine's CPU and memory. The amount of time used for local calculations is added to the time spent performing (simulated) globally visible operations to derive each thread's notion of the current time. Proteus makes sure a thread can only see global events within the scope of its local time.

3.2 Index Distribution Benchmark

Index-distribution is the simple algorithm of making a request and waiting some time before the request is repeated. In this case, the amount of time between requests is randomly chosen between 0 and `work`, a constant that determines the amount of load present. The value `work` = 0 represents the familiar counting benchmark, providing the highest possible load for the number of processors given. A higher value, usually `work` = 1000 is chosen to better distribute the requests over time, providing a lower-load environment. We ran this benchmark for a fixed amount of time on the Alewife machine (10^7 cycles), varying the number of processors[5] and the value of `work`. We also ran this benchmark on the Proteus simulator (10^5 cycles), and correlated the results. Since there are usually startup costs, the algorithms were run for some fixed time before the timing was started.

We mainly collected throughput data. The throughput is the total number of `get_next_index()` operations that completed in the time allowed. We also examined latency, the average amount of time between the call to `get_next_index()`

[5]Throughout this paper, each processor only runs one process

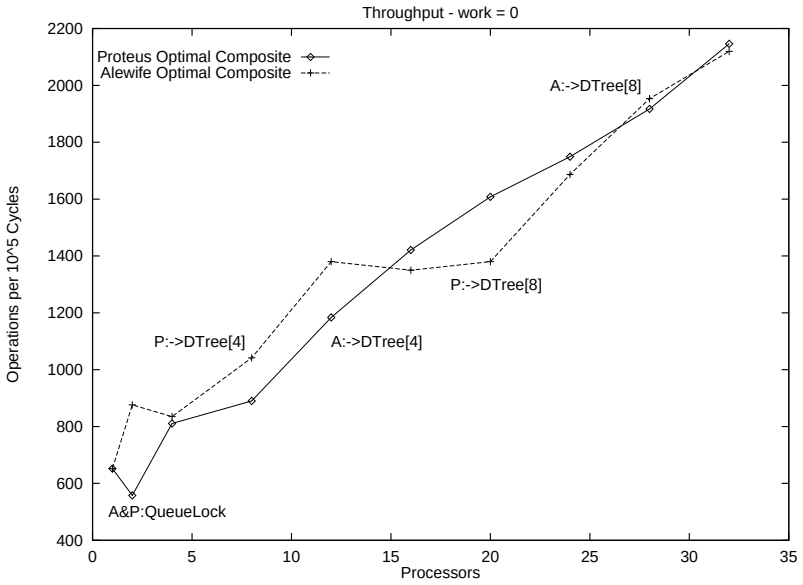

Fig. 18. Optimal Composite on Proteus and normalized Alewife

and its completion, but these numbers are clearly related and one can be calculated from the other.

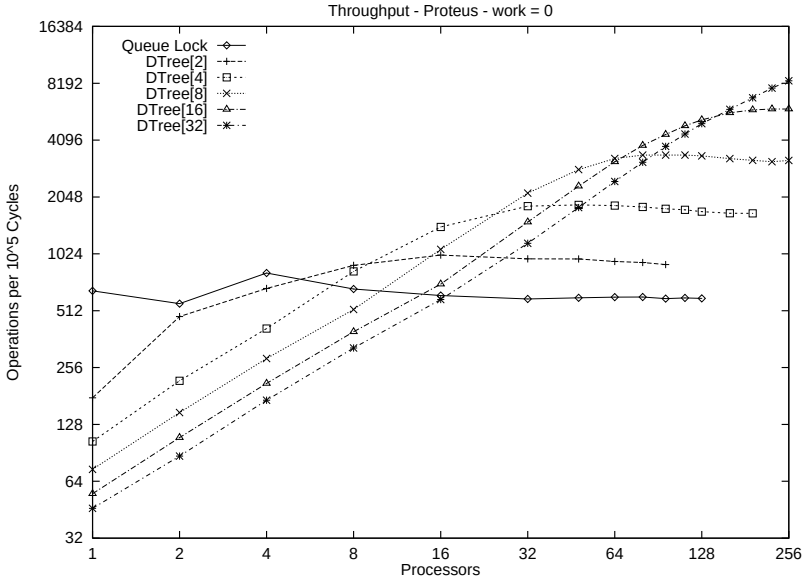

Fig. 19. Throughput of Diffracting Trees and Queue Lock on Proteus

The algorithms we ran were the Reactive Diffracting Tree, Diffracting Trees of widths 2, 4, and 8 (and on Proteus, 16 and 32), and an a queue-lock based counter.

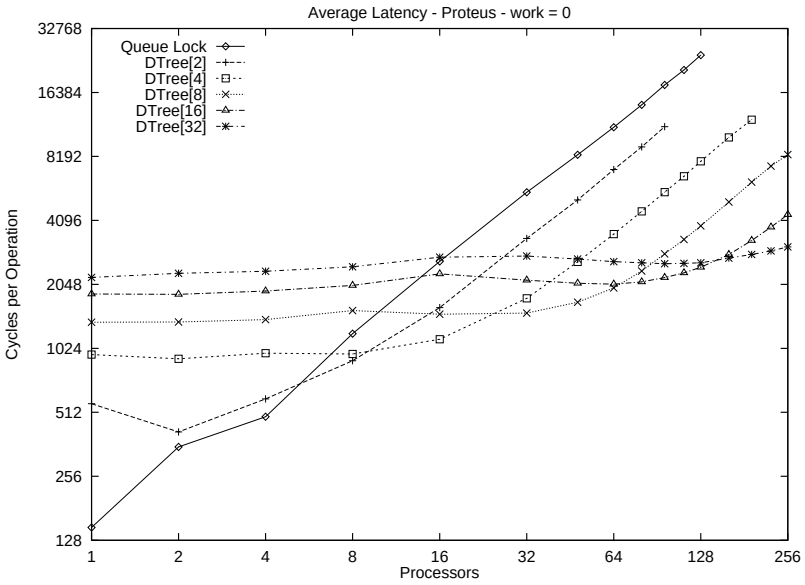

Fig. 20. Latency of Diffracting Trees and Queue Lock on Proteus

In this and all other benchmarks in this paper we used the following data structure implementations.

The queue-locks we used were the MCS queue locks of Mellor-Crummey and Scott [25]. The MCS queue-lock consists of a queue in the form of a linked-list of processors, each pointing towards its successor, waiting for its predecessor to wake it up once it is done with the lock. There is a tail pointer which directs new processors to the end of the queue. The queue-lock code was implemented using an atomic register-to-memory-swap operation.

The Diffracting Tree algorithm we used is the improved Diffracting Tree algorithm of [31]. We used queue-locks on the counters at the leaves of the Diffracting Tree as opposed to spin-locks in the original implementation of [32]. A detailed empirical study of the effects of using queue-locks can be found in [10].

The Steady-State analysis of [31] suggests that there should be $cd2^d$ prism locations in the tree, with $c2^d$ locations on each level of the tree, where c is a constant. We experimented by comparing trees at each level and found that $c = 1/2$ was the best factor overall.

3.2.1 *Alewife Results.* This section presents the first performance results for Diffracting Trees on the Alewife machine. [6] For the Reactive Diffracting Tree, we set the number of consecutive timings before a change to be 80, a good experimental number that limited the number of oscillations. Our experiments also

[6] in fact, for any counting networks structure, as the experiments of Herlihy et. al [14] were conducted on the ASIM Alewife simulator.

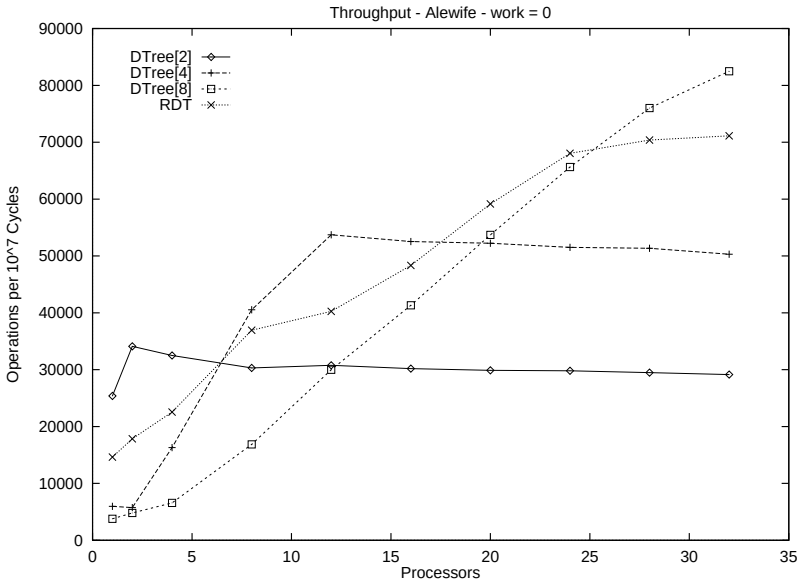

Fig. 21. Alewife Throughput for Diffracting Trees, Queue-Lock Based Counter, and RDT

determined that the best fold and unfold threshold times were 150 and 800 cycles. Figure 21 shows throughputs for a queue-lock based counter, diffracting trees of depth 1, 2, and 3, and the RDT. The most interesting result is that the RDT surpasses all of the diffracting trees shown for a brief range. This is due to its ability to

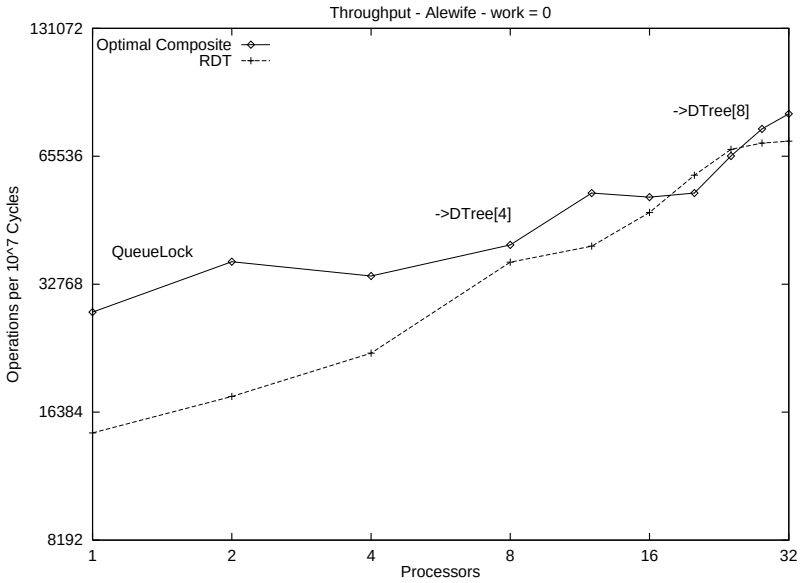

Fig. 22. Optimal Composite vs. RDT

expand only where needed, supplying irregularly sized trees which perform better in this range.

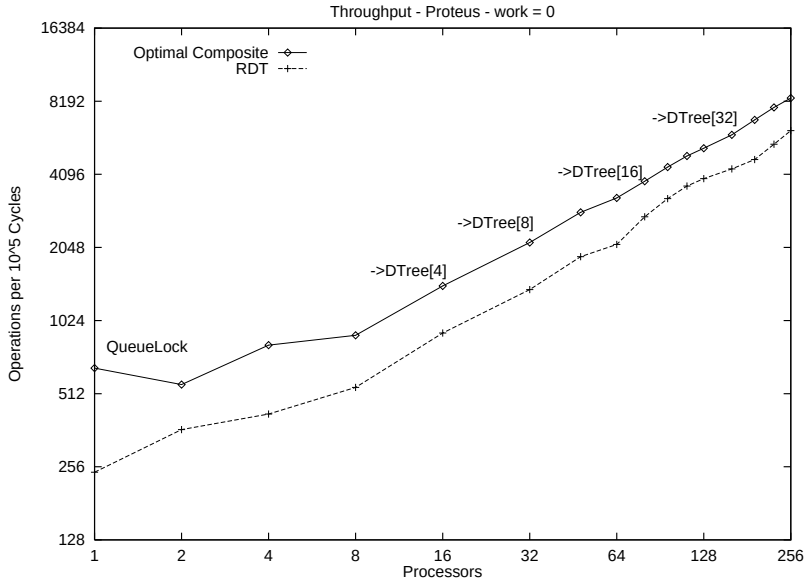

Fig. 23. Throughputs of Optimal Composite vs. RDT on Proteus under high load

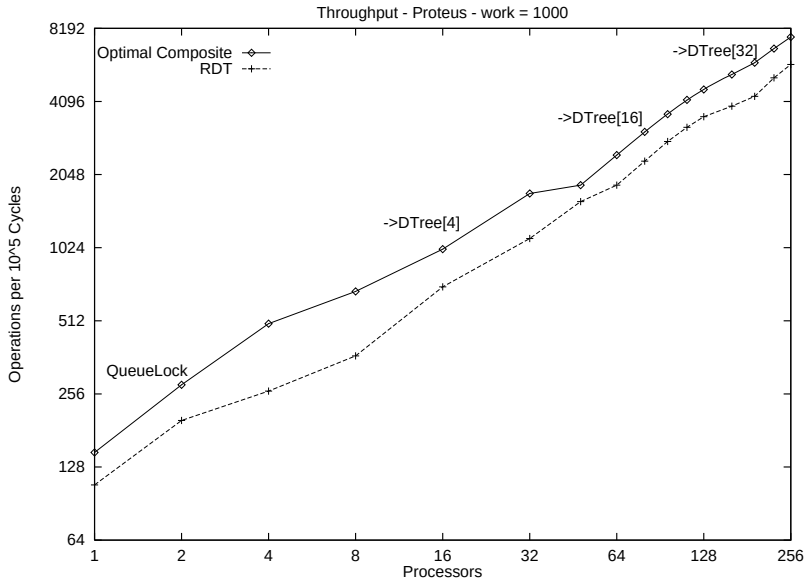

Fig. 24. Throughputs of Optimal Composite vs. RDT on Proteus under low load

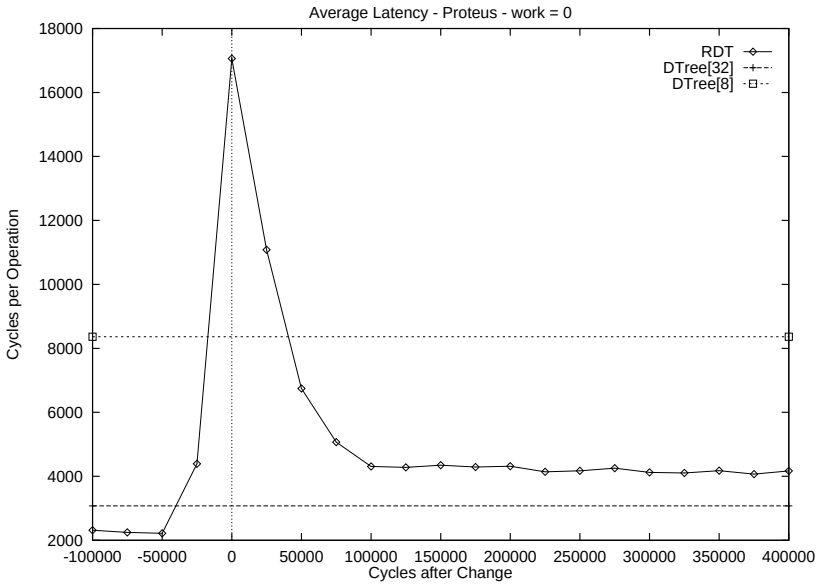

Fig. 25. Average latency of RDT over time in response to sudden surge

Since the Reactive Diffracting Tree should represent the optimal diffracting trees at their peak performance points, we constructed a composite graph of the diffracting tree and queue-lock counter throughputs, with the highest throughput from any diffracting tree or queue-lock counter at a given load level chosen for the graph. We show the optimal composite vs. the RDT for the Alewife in Figure 22 under high load. The throughput and latency appear to stay within a factor throughout its performance. The average ratio between the throughput of the RDT and the optimal composite is 1.27.

3.2.2 *RDT Results on Proteus.* Unfortunately, the Alewife machine only has 32 nodes. Since larger Alewife versions are not available, we relied on simulations to provide higher load results. We used the Proteus simulator to simulate the Alewife machine, although our simulation does not fully implement Alewife's LimitLESS cache-coherence policy.

It is thus important to compare the results gathered on the Alewife with the same benchmarks on Proteus, to make sure that the results can be carried over. Figure 17 is the counterpart to Figure 21. Notice that the shapes of the Diffracting Trees look similar, although they seem to flatten out more quickly on the Alewife than on the Proteus. But, we really need to see two curves side by side. We construct a Proteus optimal composite for throughput for 1 to 32 processors and normalize the Alewife curve to it. This graph is shown in Figure 18. The results show that the Alewife trees have a higher optimal load level, but the graphs still look comparable, a good result for Proteus.

We now extend the Proteus results up to 256 processes, and add Diffracting Trees of depths 4 and 5. Figure 19 shows the throughputs and latencies of Diffracting

```
producer:
    repeat
        produce(val);
        wait until the element is consumed;
    until a total of 2560 elements are consumed

consumer:
    repeat
        consume()
    until a total of 2560 elements are consumed
```

Fig. 26. Code for Producer/Consumer

Trees of depth 0 (queue-lock based counter) through 5. The Proteus environment is different enough to require a change in some of the constants. The difference in the timing mechanisms forced us to move the fold threshold up to 200 cycles. However, the queue-locks had more stable waiting times, enabling us to bring the consecutive timings threshold down to 25.

We show the comparison between the optimal composite and the RDT in Figure 23 for high load case ($work = 0$) and in Figure 24 for low load case ($work = 1000$). The results showed that Proteus charged more for the overhead required in computing the changes, but this seems to be a constant factor that is machine-dependent. This could be attributed to the cache-coherence differences between the two architectures. For the high load case, the average ratio between the two throughputs was 1.56, and in the low load case, the average factor was 1.41.

3.3 Large Load-Change Benchmark

We measured the response of a RDT to a sudden spike in load levels, measuring the average latency of the RDT in fixed width intervals before and after the change occurred, and graphing the change in the average latency over time. Here, the system constant for the number of consecutive timings was set at 10 to better handle sudden changes.

We ran the index-distribution benchmark with 32 participating processes for a fixed amount of time and `work` = 0, to allow the tree to best fit the load. The tree sized to a depth 3 tree. We then started timing for four time intervals of 25,000 cycles, and allowed an increase in the number of processors to 256, timing for 400,000 additional cycles. The tree grew to depth 5. Figure 25 shows the plot of these measurements. As you can see, it takes about 100,000 cycles for the curve to level off, which given an eventual average latency of 4,000 cycles, indicates that it took about 25 equivalent passes through the tree to expand 2 levels, which is what would be expected with the consecutive timings constant set at 10. The throughput before the change occurred was around 340 operations per 25,000 cycles. At the top of the spike, the throughput goes up to around 440 operations, and as the latency drops off, the throughput rises quickly to 1500 operations and remains steady.

The plot also contains Diffracting Trees of depth 3 and 5 with their average latency at 256 processors, which are what the RDT emulates before and after the change. Here is a good example of the tradeoff that a developer must consider in choosing to use the RDT. Imagine that the developer initially used the diffracting tree of depth 3. The triangle on the left formed by the RDT and the depth 3 Diffracting Tree represents the spike in latency that the algorithm must necessarily absorb in order to change, and is a loss to the developer. However, the quadrilateral-like shape formed between the RDT and the Diffracting Tree of depth 3 to the right of the triangle is the region that a developer gains in using the RDT. Of course, the developer could choose to use the depth 5 tree all along, but the RDT outperforms this tree in the lower load case, which may usually be the common case.

3.4 Producer/Consumer Benchmarks

Job pools are data structures that store a collection of jobs that need to be performed by the a collection of processors. Any processor can enqueue (produce) a new job into the pool or dequeue (consume) a job in order to perform it. The shared counter implementation of a job pool consists of two shared counters and an array of locations, where each location can in turn be a local queue [30] protected by a *full/empty* bit [1] or a spin-lock. To enqueue a job, a processor requests a value from the producer counter and places the job at the corresponding location (the index modulo the array width) in the array, then sets the full/empty bit to *full*. To dequeue a job, a processor requests a value from the consumer counter and goes to find a job in the appropriately indexed location. If the location is empty it waits until it is *full*, then removes an entry from that location, and if it is now empty it sets the locations bit to *empty*.

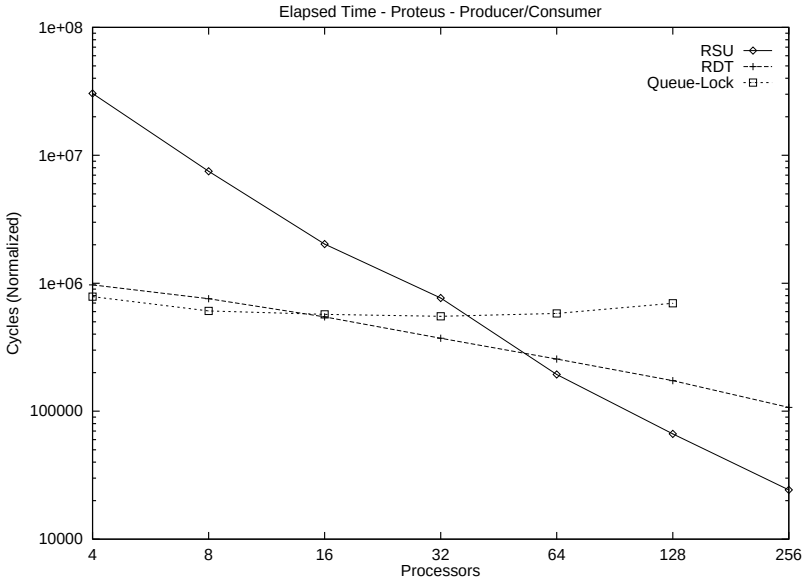

Fig. 27. Producer/Consumer Performance

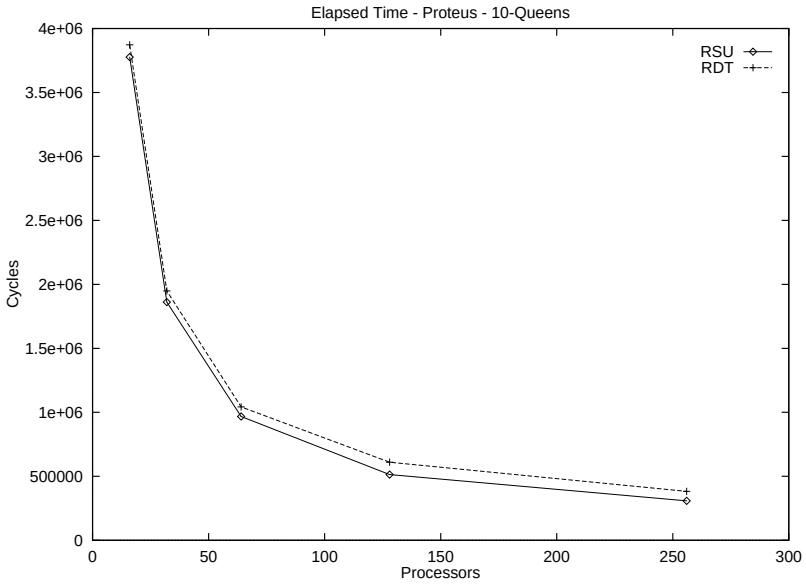

Fig. 28. 10-Queens Performance

An alternative job pool scheme consists of one of many randomized load balancing techniques. Here, processors keep local job queues from which they choose jobs to execute, and participate in load balancing to best distribute their job allocations. We compare the RDT based job pool to the randomized load balancing scheme of Rudolph et. al(RSU) [28], which we chose as a representative of this class of algorithms (though one could alternately have chosen an algorithm like the job-stealing scheme of Blumofe and Leiserson [5]). In RSU, a processor about to dequeue a job attempts to load balance with probability inversely proportional to the size of its job queue. If it decides to load balance, it picks a processor at random and attempts to equalize their job queue sizes.

In high load situations where processors frequently enqueue and dequeue jobs, randomized load balancing algorithms currently outperform shared structures such as diffracting trees [30]. The lock-based counters do well against RSU in the low load levels, and the distributed counters seem to come close to RSU's level of performance, but overall, no shared structure has been able to effectively compete with RSU. We now show that the RDT is an effective competitor.

3.4.1 *10-Queens.* The n-Queens problem is a classical test benchmark for job-queue style algorithms. In this benchmark, which mimics the behavior of an algorithm for solving the classical n-queens chess problem, every consume operation produces 10 new jobs at a higher depth until a limit is hit. The recursive nature of the algorithm leads it to apply different load levels on the producer and consumer functions. Under low loads, the counters can become lock-based algorithms and compete effectively against RSU. As the number of processors participating increases, the trees can grow larger to give the distributed performance necessary

```
Initialization
    produce one instance with depth=0
repeat
    instance = consume();
    wait 8000 cycles;

    if instance's depth < 3 then
      produce 10 instances with depth greater by 1
until all instances have been consumed
```

Fig. 29. Code for 10-Queens

to compete with RSU. Figure 28 shows how close the diffracting tree comes close to RSU in total time elapsed throughout the differing load levels.

3.4.2 *Sparse Producer/Consumer Actions.* The pitfall of RSU and the other randomized load balancing algorithms is the poor performance that occurs under sparse access patterns. To exhibit this, we make half the active processors consumers and the other half producers. Producers initially produce a job and wait until that job is consumed before they produce a new job. This continues until a total of 2560 jobs have been completed. This creates a sparse access pattern in the system since any load balancing transaction could at most shift one job, which is the necessary consumption for the production to continue. We ran this system for RSU, a RDT based job pool, and a centralized job pool protected by a queue-lock. We measure the time elapsed between the beginning of the benchmark until 2560 elements are consumed, and show the results in Figure 27. As one can see, the RDT provides near queue-lock performance in low-loads, and approaches the performance of RSU in higher loads.

4. CONCLUSION

Our work on a reactive diffracting tree structure was inspired by Lim and Agarwal's reactive lock constructions[20; 21]. Following our work, Shavit and Zemach [33] have recently presented combining funnels, a reactive data structure based on the combining paradigm [12; 11]. We hope these new structures will encourage other researchers to adopt the reactive approach in designing scalable data structures for multiprocessor machines. Below are some examples of further work to be done.

(1) One should attempt to improve the current implementation by designing a single lock version of the algorithm (locking one node only to perform folding or unfolding). This will require overcoming the complexity of keeping the output values from the tree consistent. Once this step is taken, the RDT will be closer to being wait-free [16]: the `state` variable will still need to be checked during the counters fetch-and-increment operation, but a hardware based conditional-fetch-and-increment would be enough to make this algorithm wait-free.
(2) One should implement a message passing version of the RDT algorithm. In a message passing system, different processors act as nodes in the tree, passing

messages to other nodes as a substitute for traversing the tree. This allows a processor to solely control a node, providing better ability to estimate the load levels and accurately decide when to grow or shrink.

(3) One should design a reactive elimination tree, a form of the diffracting tree presented by Shavit and Touitou [29], that supports both enqueue and dequeue operations on structures such as pools or stacks. Unlike a Diffracting tree, and elimination tree allows both tokens and anti-tokens, allowing them to be eliminated if they collide. This collision corresponds to an enqueue and a dequeue exchanging values locally, and greatly enhances performance since collided operations do not need to traverse the tree. It would be interesting to design folding and unfolding mechanisms that preserve global consistency given that there are both tokens and anti-tokens concurrently in the data-structure.

REFERENCES

[1] A. Agarwal, D. Chaiken, K. Johnson, D. Krantz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Maa, and D. Nussbaum. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication. Also, appears as MIT Technical Report MIT/LCS/TM-454, June 1991.

[2] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 396–406, May 1989.

[3] T. Anderson. The performance of spin lock alternatives for shared memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[4] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, September 1994. Earlier version in *Proceedings of the 23rd ACM Annual Symposium on Theory of Computing*, pp. 348–358, May 1991. Also, MIT Technical Report MIT/LCS/TM-451, June 1991.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Santa Fe, New Mexico, 20–22 November 1994. IEEE.

[6] E. A. Brewer, , and C. N. Dellarocas. Proteus user documentation, version 4.0, March 1992.

[7] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. Proteus: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, September 1991.

[8] Costas Busch and Marios Mavronicolas. A logarithmic depth counting network (abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, page 274, Ottawa, Ontario, Canada, 2–23 August 1995.

[9] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *asplosIV*, pages 224–234, Santa Clara, California, 1991.

[10] G. Della-Libera. Dynamic diffracting trees. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, July 1996.

[11] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficent synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, Boston, Massachusetts, April 1989.

[12] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer designing an MIMD parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1984.

[13] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–70, June 1990.

[14] M. Herlihy, B. H. Lim, and N. Shavit. Scalable concurrent counting. *ACM Transactions on Computer Systems*, 13(4):343–364, November 1995.

[15] M. Herlihy, N. Shavit, and O. Waarts. Low contention linearizable counting. In *Proceedings of the 32nd Annual Symposium on the Foundations of Computer Science (FOCS)*, pages 526–535, San Juan, Puerto Rico, October 1991. IEEE. Detailed version with empirical results appeared as MIT Technical Memo MIT/LCS/TM-459, November 1991.

[16] M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[17] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[18] M. Klugerman. *Small-Depth Counting Networks*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA 02139, 1994.

[19] M. Klugerman and C. G. Plaxton. Small-depth counting networks. In *Proceedings of the 24th ACM Symposium on Theory of Computing (STOC)*, pages 417–428, 1992.

[20] B. H. Lim. *Reactive Synchronization Algorithms for Multiprocessors*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technolgy, February 1995.

[21] B. H. Lim and A. Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*,, pages 25–35, October 1994.

[22] Nancy A. Lynch, Nir Shavit, Alexander A. Shvartsman, and Dan Touitou. Counting networks are practically linearizable. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 280–289, Philadelphia, Pennsylvania, USA, 23–26 May 1996.

[23] M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for on-line problems. volume 20 of *Proceedings of the Symposium on Theory on Computing*, pages 322–333, 1988.

[24] Marios Mavronicolas, M. Papatriantafilou, and P. Tsigas. The impact of timing on lineraizability in counting networks. In *Proceedings of the Eleventh International Parallel Processing Symposium*, pages 684–688, May August 1996.

[25] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems, TOCS*, 9(1):21–65, February 1991. Earlier version published as TR 342, University of Rochester, Computer Science Department, April 1990, and COMP TR90-114, Center for Research on Parallel Computation, Rice UNIV, May 1990.

[26] G. Pfister and V. Norton. "hot spot" contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, 1985.

[27] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proceedings of the 11th Annual Symposium on Computer Architecture*, pages 340–347, June 1984.

[28] L. Rudolph, M. Slivkin, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, July 1991.

[29] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. In *SPAA'95: 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 54–63, Santa Barbara, California, July 1995. Also, Tel-Aviv University *Technical Report*, January 1995.

[30] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. *Theory of Computing Systems*, 30(6):645–670, November/December 1997.

[31] N. Shavit, E. Upfal, and A. Zemach. A steady state analysis of diffracting trees. *Theory of Computing Systems*, 31(4):403–423, July/August 1998.

[32] N. Shavit and A. Zemach. Diffracting trees. *ACM Transactions on Computer Systems, TOCS*, 14(4):385–428, November 1996.

[33] N. Shavit and A. Zemach. Combining funnels. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, pages 61–70, Puerto Vallarta, Mexico, June 28th – July 2nd 1998.

[34] P. C Yew, N. F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, pages 388–395, April 1987.