Theory Comput. Systems **30**, 645–670 (1997)

Theory of Computing Systems
© 1997 Springer-Verlag New York Inc.

Elimination Trees and the Construction of Pools and Stacks*

N. Shavit[1,2] and D. Touitou[1]

[1]Computer Science Department, Tel Aviv University,
Ramat Aviv 69 978, Israel
{shanir,danidin}@cs.tau.ac.il

[2]Laboratory for Computer Science, MIT,
545 Technology Square, Cambridge, MA 02139, USA
shanir@theory.lcs.mit.edu

Abstract. Shared *pools* and *stacks* are two coordination structures with a history of applications ranging from simple producer/consumer buffers to job-schedulers and procedure stacks. This paper introduces *elimination trees*, a novel form of diffracting trees that offer pool and stack implementations with superior response (on average constant) under high loads, while guaranteeing logarithmic time "deterministic" termination under sparse request patterns.

1. Introduction

As multiprocessing breaks away from its traditional number crunching role, we are likely to see a growing need for highly distributed and parallel coordination structures. A real-time application such as a system of sensors and actuators will require fast response under both sparse and intense activity levels (typical examples could be a radar tracking system or a traffic flow controller). Shared *pools* offer a potential solution to such coordination problems, with a history of applications ranging from simple producer/consumer buffers to job-schedulers [6] and procedure stacks [26]. A *pool* [19] (also called a pile [22], global pool [6], or a producer/consumer buffer) is a concurrent data-type which supports the abstract operations: `enqueue(`e`)`—adds element e to the pool, and `dequeue`—deletes and returns some element e from the pool. A stack is a pool with a last-in-first-out (*LIFO*) ordering on enqueue and dequeue operations.

* A preliminary version of this paper appeared in the *Proceedings of the* 7*th Annual Symposium on Parallel Algorithms and Architectures* (*SPAA*), pages 54–63, July 1995. This research was supported by Contracts HRPA #F19628-95-C-0118, AFOSR/ONR #F49620-94-1-0199, NSF #9225124-CCR, and #9520298-CCR.

Since the formal introduction of the problem and its first solution by Manber [19], the literature has offered us a variety of possible pool implementations. On the one hand there are queue-lock-based solutions such as those of Anderson [4] and Mellor-Crummey and Scott [20], which offer good performance under sparse access patterns, but scale poorly since they offer little or no potential for parallelism in high-load situations. On the other hand there are a variety of "load-balanced local pools"-based algorithms like Manber's *search tree* structure [19] and the simple and effective randomized *work-pile* and *job-stealing* techniques as designed by Kotz and Ellis [15], Rudolph *et al*. [22], Lüling and Monien [17], and Blumofe and Leiserson [6]. These algorithms offer good *expected* response time under high loads, but very poor performance as access patterns become sparse (their expected response time becomes linear in n—the number of processors in the system—as opposed to that of a "deterministic" queue-lock-based pool that is linear in the number of participating processors). This linear behavior under sparse access patterns also holds for Manber's tree-based deterministic *job-stealing* method [19].

Shavit and Zemach's *diffracting trees* [25] have recently been proposed as a reasonable middle-of-the-road solution to the problem. Width w trees guarantee termination within $O(\log w)$ time (where $w \ll n$) under sparse access patterns, and rather surprisingly manage to maintain similar average response time under heavy loads.

1.1. *Elimination Trees*

This paper introduces *elimination trees*, a novel form of diffracting trees that offers pool implementations with the same $O(\log w)$ termination guarantee under sparse patterns, but with a far superior response (on average constant) under high loads. Our empirical results show that unlike diffracting trees, and in spite of the fact that elimination trees offer a "deterministic" guarantee of coordination,[1] they scale like "randomized" methods [6], [15], [17], [22], providing improved response time as the load on them increases.

In a manner similar to diffracting trees, elimination trees are shared data structures constructed from simple one-input two-output computing elements called *elimination balancers* that are connected to one another by wires to form a balanced binary tree with a single root input wire and multiple leaf output wires. While diffracting trees route *tokens*, elimination trees route both *tokens* and *antitokens*. These arrive on the balancer's input wire at arbitrary times, and are output on its output wires. The balancer acts as a toggle mechanism, sending tokens and antitokens left and right in a balanced manner. For example, to create a pool implementation that has stack-like behavior, the balancer can consist of a single bit, with the rule that tokens toggle the bit and go to the 0 or 1 output wire according to its *old* value, while antitokens toggle the bit and go left or right according to its *new* value. Now, imagine that stack array entries are placed at the leaves of the tree, and think of tokens as enqueue ("push") requests and antitokens as dequeue ("pop") requests. Figure 1 shows a width four tree after three enqueues and a dequeue have completed. The reader is urged to try this sequence with toggles initially 0. The state of the balancers after the sequence is such that if a token will enter next it will see 0 and then 1 and end up on wire y_2, while if an antitoken is next to enter it will get a 1 and then a 0 and end up on wire y_1, finding the value to be deleted. In fact, our

[1] They guarantee that a dequeue operation on a nonempty queue will always succeed.

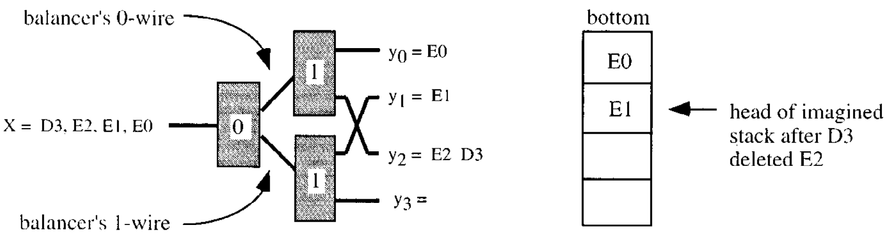


Fig. 1. A sequential execution on a STACK[4] elimination tree.

tree construction is a novel form of a *counting-network*-based [5] counter, that allows decrement (antitoken) operations in addition to standard increment (token) operations.

However, this simple approach is bound to fail since the toggle bit at the root of the tree will be subject to multiple concurrent modification requests that can be answered only one at a time. It will thus become a hot-spot [11], [21] and a sequential bottleneck, making our solution no better than a centralized stack implementation. The problem is overcome by placing a *diffracting prism* [25] structure in front of the toggle bit inside every balancer (see Figure 2). Pairs of tokens attempt to "collide" on independent locations in the prism, diffracting in a coordinated manner, one to the 0-wire and one to the 1-wire, thus leaving the balancer without ever having to toggle the shared bit. This is not a problem since in any case after both toggled it, the bit would return to its initial state. This bit will only be accessed by processors that did not succeed in colliding, and they will toggle it and be directed as before.

Our first observation is that the stack behavior will not be violated if pairs of antitokens, not only tokens, are diffracted. The second, more important, fact is that it will continue to work if collisions among a token and an antitoken result in the "elimination" of the pair, without requiring them to continue traversing the tree! In other words, a token and antitoken that meet on a prism location in a balancer can exchange enqueue/dequeue information and complete their operation without having to continue through $\log w$ balancers. In fact, our empirical tests show that under high loads, most tokens and antitokens are eliminated within two levels. Of course, the tree structure is needed since one could still have long sequences of enqueues only.

We compared the performance of elimination trees with other known methods using the Proteus Parallel Hardware Simulator [7] in a shared memory architecture similar to the Alewife machine of Agarwal *et al.* [1]. We first compared, under high loads, a variety of methods that can be used to implement a stack-like pool and are known to perform well under sparse access patterns. We found that elimination trees scale *substantially* better than all of these methods, including queue-locks [20], combining trees [12], and diffracting trees [25].

We then compared elimination trees with the *load-balanced local pools* techniques [19], [15], [22], [17], [6] which cannot be used to implement a stack-like pool and theoretically provide only linear performance under sparse access patterns. We found that in many high-load situations elimination trees are inferior to these methods (as is explained in what follows, we chose for the comparison a representative technique, the randomized technique of Rudolph *et al.* [22]), especially for job distribution applications where a typical processor is the dequeuer of its latest enqueue (though in many cases not by much). However, our empirical evidence suggests that elimination trees provide a better response time, up to a factor of 30 faster than the randomized methods under sparse loads. Finally, we present evidence that our new elimination balancer design offers a more scalable diffracting balancer construction even in cases where no collisions are possible.

In summary, this paper introduces *elimination trees*, novel forms of diffracting trees that we beleive will offer effective implementations concurrent pool structures. The remainder of this paper is organized as follows. In Section 2 we present a concurrent pool specification, our pool implementation, and its empirical evaluation. Section 3 does the same for our stack-like pool structure. Then in Section 4 we discuss future research directions.

2. Pools

We begin with our pool specification and implementations, later showing how to modify them to create stack-like pools.

A *pool* [19] (also called a pile [22], centralized "pool" [6], or a producer/consumer buffer) is a concurrent data-type which maintains a multiset of values by supporting the abstract operations: `enqueue(`e`)`—adds element e to the multiset, and `dequeue`—deletes and returns some element e from the multiset. For simplicity, assume that all enqueued elements e are unique, that is, multiset is simply a set. A pool is a relaxation of a first-in-first-out queue: apart from the queue's basic safety properties, no causal order is imposed on the enqueued and dequeued values. However, it is required that:

(P1) an enqueue operation always succeeds, and
(P2) a dequeue operation succeeds if the pool is nonempty, that is, for every execution in which the number of enqueue operations is greater than or equal to the number of dequeue operations, all the dequeue operations succeed.

A *successful* operation is one that is guaranteed to return an answer within finite (in our construction, *bounded*) time. Note that the randomized decentralized techniques of [6], [15], [17], and [22] implement a weaker "probabilistic" pool definition, where condition (P2) is replaced by a *probabilistic* guarantee that dequeue operations succeed.

2.1. *Elimination Trees*

Our pool implementation is based on the abstract notion of an *elimination tree*, a special form of the diffracting tree data structures introduced by Shavit and Zemach in [25]. Our formal model follows that of Aspnes *et al*. [5] and the I/O-automata of Lynch and Tuttle [18].

An *elimination balancer* is a routing element with one input wire x and two output wires y_0 and y_1. *Tokens* and *antitokens* arrive on the balancer's input wire at arbitrary times, and are output on its output wires. Every token carries a value. Whenever a token "meets" an antitoken in a balancer, it passes the value to the antitoken and both token and antitoken are "eliminated"—effectively never leaving the balancer. More formally, a pool balancer is a shared object that allows processors to execute `TokenTraverse(TokenType, v)` operations which have as inputs the token's type, `TOKEN` or `ANTITOKEN`, and its value `v` (which is nonempty in case of a `TOKEN` type traversal). Each such operation returns `0` or `1`, depending on which of the output wires y_0 and y_1 the token should proceed, or the pair `(ELIMINATED,v)` meaning that the token (or antitoken) was eliminated and that the value `v` was exchanged. We slightly abuse our notation and denote by x and $\bar{x}$ the number of tokens and antitokens ever received, and by y_i and $\bar{y}_i$, $i \in \{0, 1\}$, the number of tokens and antitokens ever output on the ith output wire. The pool balancer object must guarantee:

Quiescence Given a finite number of input tokens and antitokens, the balancer will reach a *quiescent* state, that is, a state in which all the tokens and antitokens traversal operation executions have completed.

Pairing In any quiescent state, there exists a *perfect matching* between eliminated tokens and eliminated antitokens, such that the value returned by an eliminated antitoken is matched with the value carried by its corresponding eliminated token.

Pool Balancing In any quiescent state, if $x \geq \bar{x}$, then, for every output wire $i \in \{0, 1\}$, $y_i \geq \bar{y}_i$.

Let Pool$[w]$ be a binary tree of elimination balancers with a root input wire x and w designated output wires: $y_0, y_1, \ldots, y_{w-1}$, constructed inductively by connecting the outputs of an elimination balancer to two Pool$[w/2]$ trees. From the quiescence property of the balancers, given a finite number of input tokens and antitokens, Pool$[w]$ will reach a quiescence state in which all the tokens and antitokens are either eliminated or have exited through one of the Pool$[w]$ outputs. We extend pool-balancing to trees in the natural way claiming that:

Lemma 2.1. *The outputs $y_0, \ldots, y_{w-1}$ of* Pool$[w]$ *satisfy the pool-balancing property in any quiescent state.*

Proof. The proof is by induction on w. When $w = 2$ this follows directly from the balancer definition. We assume the claim for Pool$[w/2]$ and prove it for Pool$[w]$. If the number of tokens entering the root balancer of Pool$[w]$ is greater than or equal to the number of antitokens, then, by definition, this property is kept on the output wires of the root balancer, and by the induction hypothesis holds for the output wires of both Pool$[w/2]$ trees. □

On a shared memory multiprocessor, an elimination tree can be implemented 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 can run a program that repeatedly traverses the data structure from the root input pointer to some output pointer, each time shepherding a new "token" or "antitoken" through the network (see Figure 3). Constructing a *pool* object from a Pool$[w]$ tree is straightforward: each tree output wire is connected to a sequentially accessed "local" pool, a simple queue protected by a Mellor-Crummey and Scott MCS-queue-lock [20] will do. The MCS-queue-lock has the property of being "fair," and so every access request to the queue will be granted within a bounded number of operations. A process performs an enqueue operation by shepherding a token "carrying" the value down the tree. If the token reaches the output wire, the associated value is enqueued in the local pool connected to that wire. The dequeue operation is similarly implemented by carrying an antitoken through the network. If this antitoken collides with a token in a balancer, the dequeuing process returns the token's value. Otherwise it exits on a wire and performs a dequeue operation on the antitoken's local pool. Naturally if the local pool is empty the dequeuing process waits until the pool is filled and then access it. The elimination tree is thus a load-balanced *coordination* medium among a distributed collection of pools. It differs from the elegant randomized constructions of [6], [15], [17], and [22] in its deterministic dequeue termination guarantee and in performance. While work in an individual balancer is relatively high, each enqueue or dequeue request passes at most log w balancers under both high and low loads.

Theorem 2.2. *The elimination-tree-based pool construction is a correct pool implementation.*

Proof. The basic safety properties of the pool are satisfied thanks to the perfect matching between eliminated tokens. By the quiescence property of the balancers all the tokens and antitokens will eventually reach the exits of the elimination tree. Since the MCS-queue-locks controlling access to the local pools are fair, all the enqueue operations will succeed in adding their value to the local pools within some bounded number of operations and property (P1) will be satisfied. Now, if the number of dequeue operations is greater than the number of enqueue operations, by Lemma 2.1 this will eventually be the case at each of the local pools at the leaves, then no dequeue operation will ever have to wait indefinitely at a leaf. This satisfies property (P2). □

2.2. *Pool Elimination Balancers*

The scalable performance of our pool constructions depends on providing an efficient implementation of an elimination balancer.

Diffracting balancers were introduced in [25]. Our shared memory construction of a diffracting elimination balancer, apart from providing a mechanism for token/antitoken elimination, also improves on the performance of the original diffracting balancer design. While a regular diffracting balancer [25] is constructed from a single prism array and a toggle bit, the elimination balancer we use in our pool construction (see the top of Figure 2) has a sequence of prism arrays and two toggle bits, one for tokens and one for antitokens.[2] Each of the toggle bit locations is protected by an MCS-queue-lock [20]. A process shepherding a token or antitoken through the balancer decides on which wire to exit according to the value of the respective token or antitoken toggle bit, 0 to the left and 1 to the right, toggling the bit as it leaves. The toggle bits effectively balance the number of tokens (resp. antitokens) on the two output wires, so that there is in any quiescent state at most one token (resp. antitoken) more on the 0 output wire than on the 1 wire. Readers can easily convince themselves that this suffices to guarantee the pool-balancing property. However, if many tokens were to attempt to access the same toggle bit concurrently, the

[2] The two separate toggle locations are an artifact of the pool-balancing property. In our stack construction in Section 3 the elimination balancer uses a single toggle bit for both tokens and antitokens.

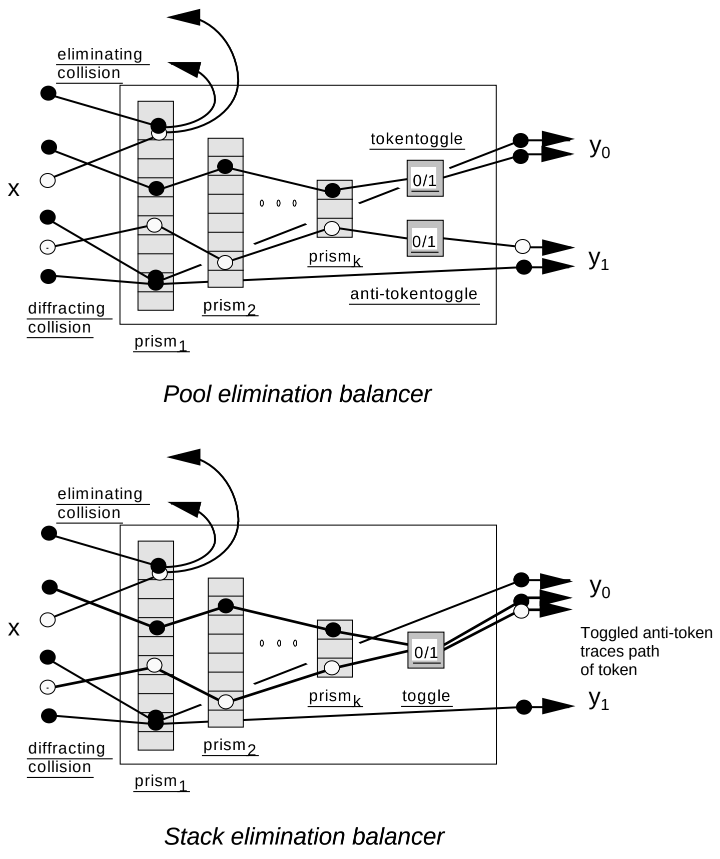


Fig. 2. The structure of pool and stack elimination balancers.

bit would quickly become a hot spot. The solution presented in [25] is to add a *prism* array in front of each toggle bit. Before accessing the bit, the process shepherding the token selects a location *l* in the prism uniformly at random, hoping to "collide" with another token which selected *l*. If a collision occurs, then the tokens "agree" among themselves that one should be "diffracted" left and the other right (the exact mechanism is described in what follows), without having to access the otherwise congested toggle bit. If such a *diffracting collision* does not occur, the process toggles the bit as above and leaves accordingly. As proved in [25], the combination of diffracted tokens and toggling tokens behaves exactly as if all tokens toggled the bit, because if any two diffracted tokens were to access the bit instead, after they both toggled it the bit state would anyhow return to its initial state. The same kind of prism could be constructed for antitokens.

The key to our new constructions is the observation that for data structures which have complementary operations (such as enqueues and dequeues), a substantial performance benefit can be gained from having a joined prism for both tokens and antitokens. In addition to toggling and diffracting of tokens and antitokens, if a collision between

```
root : global ptr to root of elimination tree

procedure enqueue(v:value);
  b:= root
  while not leaf(b)
     r :=TokenTraverse(TOKEN,v) on  balancer b;
     case r of
       ELIMINATED: return;
       0         : b := left child of b;
       1         : b := right child of b;
     endcase
  endwhile
  enqueue_local_pool(b,e)

function dequeue(): value;
  b:= root
  while not leaf(b)
     r:=TokenTraverse(ANTITOKEN,EMPTY) on balancer b;
     case r of
       <ELIMINATED,v> : return v;
       0         : b := left child of b;
       1         : b := right child of b;
     endcase
  endwhile
  return dequeue_local_pool(b);
```

Fig. 3. Tree traversal code.

a token and antitoken occurs in the shared prism, they can be "eliminated" (exchanging the complementary information among themselves) without having to continue down the tree. We call this an *eliminating collision*. Unlike with diffracting collisions, if the eliminating collision had not occurred, each of the token and antitoken toggle bits would have changed. Nevertheless, the combination of toggling, diffracting, and elimination preserves the pool-elimination balancer's correctness properties, which by Lemma 2.1 guarantees pool balancing.

The *size* of (number of locations in) the prism array has critical influence on the efficiency of the node. If it is too high, tokens will miss each other, lowering the number of successful eliminations, and causing contention on the toggle bits. If it is too low, too many processes will collide on the same prism entry, creating a hot-spot. We typically found the optimal performance was when the prism width at a balancer on a given level is the same as the width of the subtree below it (this conforms with recent projections based on steady-state analysis [24]). Moreover, unlike the single prism array of [25], we found it more effective to pass a token through a series of prisms of decreasing size, thus increasing the chances of a collision. This way, at high contention levels most of the collisions will occur on the larger prisms, while at low levels they happen on the smaller ones.

Figure 4 gives the code for traversing an elimination balancer. Note that for algorithmic simplicity we omitted input values and the code for their exchange, and have deferred a discussion of this issue to Section 2.4.

```
Location: shared array[1..NUMPROCS];

Function TokenTraverse(b: ptr to bal, mytype: TokenType)
                                              returns (ptr to bal or ELIMINATED);
   Location[mypid] := <b,mytype>;
   /* Part 1 : attempt to collide with another token on k prism levels */
   for i:=1 to k do
        place := random(1,size_i);
        him   := register_to_memory_swap(Prism_i[place],mypid);
        if not_empty(him) then
           <his_b,his_type> := Location[him];
           if his_b = b  then
              if compare_and_swap(Location[mypid],<b,mytype>, <0,EMPTY>) then
                 if my_type = his_type then
                    if compare_and_swap(Location[him],<b,his_type>,<0,DIFFRACTED>) then
1.                     return b->OutputWire[1]
                    else Location[mypid] := <b,mytype>;
                 else if compare_and_swap(Location[him],<b,his_type>,<0,ELIMINATED>) then
2:                     return ELIMINATED;
                      else Location[mypid] := <b,mytype>;
              else if  Location[mypid]= <0,DIFFRACTED> return (b->OutputWire[0])
                   else return ELIMINATED
        repeat b->Spin times /* wait in hope of being collided with */
            if  Location[mypid] = <0,DIFFRACTED>  then return b->OutputWire[0];
            if  Location[mypid] = <0,ELIMINATED> then return ELIMINATED;
   /* Part 2  access toggle the bits */
   AquireLock(b->Locks[mytype]);
   if compare_and_swap(Location[mypid],<b,my_type>, <0,EMPTY>) then
        i:= b->Toggles[mytype];
        b->Toggles[mytype] := Not(i);
        ReleaseLock(b->Locks[mytype]);
3:      return b->OutputWire[i];
   else ReleaseLock(b->Locks[mytype]);
        if  Location[mypid]= <0,DIFFRACTED> return (b->OutputWire[0])
        else return ELIMINATED
```

Fig. 4. Traversing an eliminating balancer.

Apart from reading and writing memory, our implementation uses a hardware

- `register_to_memory_swap(addr,val)` operation, and a
- `compare_and_swap(addr,old,new)`, an operation which checks if the value at address `addr` is equal to `old`, and, if so, replaces it with `new`, returning `TRUE` and otherwise `FALSE`.

Our implementation also uses standard `AquireLock` and `ReleaseLock` procedures to enter and exit the MCS-queue-lock [20].

Initially, processor *p* announces the arrival of its token at node *b*, by writing *b* and its token type to `Location`[*p*]. It then chooses a location in the $\texttt{Prism}_1$ array uniformly at random (note that randomization here is used only to load-balance processors over the prism, and could be eliminated in many cases without a significant performance penalty) and swaps its own PID for the one written there. If it read a PID of an existing processor q (i.e., `not_empty(him)`), p attempts to collide with q. This collision is accomplished by first executing a $\langle$`his_b,his_type`$\rangle$ `:= Location[him]` read operation to determine the type of token being collided with, and then performing two compare-and-swap operations on the `Location` array. The first clears p's entry, assuring no other processor will collide with it during its collision attempt (this eliminates race conditions). The second attempts to mark q's entry as "collided with p," notifying q of the collision type: `DIFFRACTED` or `ELIMINATED`. If both compare-and-swap operations succeed, the collision is successful, and p decides based on collision type either to diffract through the right output wire or to be eliminated. If the first compare-and-swap fails, it follows that some other processor r has already managed to collide with p. In that case p diffracts through the left output wire or is eliminated, depending on the type of processor that collided with it. If the first succeeds but the second fails, then the processor with whom p was trying to collide is no longer at balancer b, in which case p resets its `Location` entry to contain the balancer name and its token type, and, having failed to "collide with" another processor, spins on `Location[`p`]` waiting for another processor to "collide with it." If after `spin` time units no collision occurs, p restarts the whole process at the next level $\texttt{Prism}_2$ and so on. If p has traversed all the prism levels without colliding, it acquires the lock on the toggle bit, clears its element, toggles the bit, and releases the lock. If p's element could not be erased, it follows that p has been collided with, in which case p releases the lock without changing the bit and diffracts or is eliminated accordingly.

2.3. *Correctness Proof of Pool Balancer Implementation*

Clearly, if no diffractions and no eliminations occur during an execution, by the code all the tokens would access the toggle bits and the balancing property will be easily satisfied. Hence, in order to prove the correctness of our implementation we should focus on showing that eliminating and diffracting tokens are paired off correctly. For example, we must show that a scenario in which token T_1 diffracts with token T_2 and in which T_2 is not aware of it and still toggles the bit, will never happen. As a first step, we assume that every token in a given execution has a unique virtual ID T_p, and let the subscript p denote the PID of the process shepherding the token. We use the "$*$" notation throughout the paper to denote an unspecified value. In the following lemma we show that if some process p reads `Location[`q`] =` $\langle b,$`*`$\rangle$, then process q is currently shepherding a token through balancer b.

Lemma 2.3. *For every process p, if* `Location[`p`] =` $\langle b,$`*`$\rangle$ *then p is executing* `TokenTraverse` *on balancer b.*

Proof. Initially `Location[`p`]` $= 0$. From the algorithm it is clear that only p can write a value different from 0 as a balancer name in `Location[`p`]`. Since p always writes 0 into `Location[`p`]` (a successful `compare_and_swap`) before completing `TokenTraverse`, the claim follows. □

We now define a token T_p traversing a balancer b as a *diffracting* token if p has executed Line 1 in the algorithm and thus "leaves on output wire 1." Since, for every diffracting token T_p, p executed a successful `compare_and_swap(Location[him],` $\langle b$`,*`$\rangle$`,` $\langle$`0,DIFFRACTED`$\rangle$), we know by Lemma 2.3 that at the same time process `him` was shepherding some token T_{him} through b. We designate T_{him}, which "leaves on output wire 0" as *diffracted by* T_p. We also define a token T_p as an *eliminating* token if p executed Line 2. In a similar way as for diffracting tokens we designate the token T_{him} as *eliminated* by T_p. Finally we define a token T_p as a *toggling* token if p has executed Line 3 in the algorithm. From the flow control of the algorithm it is clear than a token cannot be both toggling and eliminating, or toggling and diffracting, or eliminating and diffracting.

In the next two lemmas we show that tokens are paired off correctly during elimination and diffraction.

Lemma 2.4. *Every token traversing a balancer b can be diffracted or eliminated by at most one other token.*

Proof. By way of contradiction. Assume that a token T_p, while traversing b, has been eliminated or diffracted by two other tokens T_q and T_r. In that case, both q and r have successfully executed `compare_and_swap(Location[`p`],` $\langle b$`,*`$\rangle$`,` $\langle$`0,*`$\rangle$`)`. It follows that p must have written $\langle b$`,*`$\rangle$ in `Location[`p`]` at least twice during the execution of the `TokenTraverse` carrying T_p through b. However, in that case `compare_and_swap(Location[`p`],` $\langle b$`,*`$\rangle$`,` $\langle$`0,EMPTY`$\rangle$`)` was successfully executed by p before writing $\langle b$`,*`$\rangle$ on `Location[`p`]` for the second time. A contradiction. □

Lemma 2.5. *A toggling, eliminating, or diffracting token T_p cannot be eliminated or diffracted by some other token T_q.*

Proof. The proof follows since q executes Lines 1, 2, or 3, or writes $\langle b$`,*`$\rangle$ on `Location[`q`]`, only after executing a successful `compare_and_swap(Location[`q`],` $\langle b$`,*`$\rangle$`,` $\langle$`0,EMPTY`$\rangle$`)`, no other process will be able to execute a successful `compare_and_swap(Location[`q`],` $\langle b$`,*`$\rangle$`,` $\langle$ `0,EMPTY`$\rangle$`)`. □

We now prove that:

Theorem 2.6. *The pool balancer implementation given in Figure* 4 *satisfies the pool-balancing property.*

Proof. Given any execution of the pool implementation, let d_1 and $\bar{d}_1$ be the number of diffracting (leaving on wire 1) tokens and antitokens respectively and let d_0 and $\bar{d}_0$ be the number of diffracted (leaving on wire 0) tokens and antitokens. We designate by e the number of eliminated and eliminating tokens and by $\bar{e}$ the number of eliminating and eliminated antitokens. Finally, let t and $\bar{t}$ be the number of toggling tokens and antitokens, respectively.

By Lemma 2.5, $x = d_0 + d_1 + e + t$ and $\bar{x} = \bar{d}_0 + \bar{d}_1 + \bar{e} + \bar{t}$. By Lemma 2.4, $\bar{e} = e$, $d_0 = d_1$, and $\bar{d}_0 = \bar{d}_1$. Now, if $x \geq \bar{x}$, then $t + d_0 + d_1 = x - e \geq \bar{x} - \bar{e} = \bar{t} + \bar{d}_0 + \bar{d}_1$. Consequently,

$$\left\lceil \frac{t + d_0 + d_1}{2} \right\rceil \geq \left\lceil \frac{\bar{t} + \bar{d}_0 + \bar{d}_1}{2} \right\rceil,$$

and since $d_0 = d_1$ and $\bar{d}_0 = \bar{d}_1$ then $\lceil t/2 \rceil + d_0 \geq \lceil \bar{t}/2 \rceil + \bar{d}_0$. Therefore $y_0 \geq \bar{y}_0$. Using the same arguments, it can be shown that $\lfloor t/2 \rfloor + d_1 \geq \lfloor \bar{t}/2 \rfloor + \bar{d}_1$ and therefore $y_1 \geq \bar{y}_1$. □

2.4. *Exchanging Values in Eliminating Collisions*

The purpose of the eliminating collisions is to allow enqueuers and dequeuers to exchange values and to leave the pool. The algorithm in Figure 4 can be easily modified to handle value exchanges: every process writes and reads from `Location[mypid]` a triplet ⟨`b,mytype,value`⟩ instead of just the pair ⟨`b,mytype`⟩. To eliminate an antitoken, a token writes ⟨`0,ELIMINATED,value`⟩ in the antitoken's `Location`. Note that it knows this is an antitoken following the preliminary ⟨`his_b,his_type`⟩ `:= Location[him]` read operation. In this way the eliminated antitoken will find this value and return it. On the other hand, an eliminating antitoken returns the value it has read from the eliminated token's `Location` entry. Since, the triplets stored in `Location` are written and updated atomically, only minor modifications are needed in the correctness proof: we just have to show that an eliminating (or eliminated) antitoken returns the value carried by the token it has eliminated (or was eliminated by). The proof of this lemma is identical to the proof of Lemma 2.3.

Lemma 2.7. *For every process p, if* `Location[`p`] =` ⟨b`,TOKEN,`v⟩, *then p is shepherding a token carrying value v on balancer b.*

We have shown in Lemmas 2.4 and 2.5 that eliminated tokens and antitokens are paired off correctly. We now prove that eliminated or eliminating antitokens exchange values in a proper way.

Lemma 2.8. *Every eliminated antitoken returns the value carried by the token that has eliminated it. Every eliminating antitoken returns the value carried by the token it has eliminated.*

Proof. Assume that T_p is an eliminated antitoken. Let T_q be the token which eliminated T_p. By the modified algorithm `compare_and_swap(Location[`p`],`⟨b`,ANTI-TOKEN,NULL`⟩`,`⟨`0,ELIMINATED,`v⟩`)` was successfully executed by q, where v is the value carried by T_q. Since only p can change the content of `Location[`p`]`, and it could not, it must have returned v.

Assume that T_q is an eliminating antitoken which returned a value v and let T_p be the token it eliminated. Process q executed `compare_and_swap(Location[`p`],`

$\langle b,$ `TOKEN`$, v\rangle, \langle$`0,ELIMINATED,NULL`$\rangle)$ successfully, and therefore, by Lemma 2.7, v must be the value carried by T_p. □

2.5. *Performance of the Elimination-Tree-Based Pool*

We evaluated the performance of our *elimination-tree*-based pool construction relative to other known methods by running a collection of benchmarks on a simulated distributed-shared-memory machine similar to the MIT *Alewife* machine [1] of Agarwal *et al*. The results presented hopefully exemplify the potential in using elimination trees, but in no way claim to be a comprehensive study of their performance.

Our simulations were performed using *Proteus*, a multiprocessor simulator developed by Brewer *et al*. [7]. 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 complete *cycle per cycle* hardware simulations. Instead, local operations (that do not interact with the parallel environment) are run uninterrupted on the simulating machine's CPU. 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.

Our simulated Alewife-like machine has 256 processors, each at a node of a Torus-shaped communication grid. Each node also contains a cache memory, a router, and a portion of the globally addressable memory. The cost of switching or wiring in the Alewife architecture is 1 cycle/packet. Each processor has a cache with 2048 lines of 8 bytes. The cache coherence is provided using a version of Chaiken's directory-based cache-coherence protocol [9].

2.5.1. *The Produce–Consume Benchmark*.

We begin by comparing under various loads *deterministic* pool constructions which are known to guarantee a good enqueue/dequeue time when the load is low (sparse access patterns). These methods are also the ones that can be modified to provide stack-like pool behavior. In the produce–consume benchmark each processor alternately enqueues a new element in the pool, dequeues a value from the pool, and then waits a random number of cycles between 0 and `Workload` (see Figure 5).

We ran this benchmark varying the number of processors (the architecture remained the same 256 node machine) participating in the simulation during 10^6 cycles, measuring: *latency*, the average amount of time spent per produce and consume operation,

```
repeat
  produce(val);
  val := consume;
  w   := random(0..Workload);
  wait w cycles;
until 10^6 cycles elapsed
```

Fig. 5. The Produce–consume benchmark.

```
Pool: array[1..N] of elements; - initially set to NULL -- N must be chosen optimally
headcounter, tailcounter:integer;  - initially set to 0

Procedure Enqueue(el:elements);
 i:= fetch_and_increment(headcounter);
 repeat
   flag:= compare_and_swap(Pool[i],NULL,el);
 until flag= TRUE;

Function Dequeue() returns elements;
 i:= fetch_and_increment(tailcounter);
 repeat
  repeat el := Pool[i] until el <> NULL;
  flag := compare_and_swap(Pool[i],el,NULL)
 until flag= TRUE;
return el;
```

Fig. 6. A pool based on a cyclic array and shared counters.

and *throughput*, the number of produce and consume operations executed during 10^6 cycles.

In preliminary tests we found that the most efficient pool implementations are attained when using shared counting to load balance and control access to a shared array (see Figure 6).

We thus realized the centralized pool in the style of [5], given in Figure 6, where the `headcounter` and `tailcounter` are implemented using two counters of the following type:

MCS The MCS-queue-lock of [20], whose response time is linear in the number of concurrent requests. Each processor locks the shared counter, increments it, and then unlocks it. The code was taken directly from the article, and implemented using atomic operations: `register_to_memory_swap` and `compare_and_swap` operations.

CTree A *Fetch&Inc* using an optimal width software combining tree following the protocol of Goodman *et al.* [12], modified according to [13]. The tree's response time is logarithmic in the maximal number of processors. Optimal width means that when n processors participate in the simulation, a tree of width $n/2$ will be used [13].

DTree A diffracting tree of width 32, using the optimized parameters of [25], whose response time is logarithmic in $w = 32$, which is smaller than the maximal number of processors. The prism `sizes` were 8, 4, 2, 2, and 1 for levels $1, \ldots, 5$, respectively. The `spin` is equal to 32, 16, 8, 4, and 2 for balancers at depths 0, 1, 2, 3, 4, and 5, respectively.

and compared it with:

ETree A POOL[32] elimination-tree-based pool, whose response time is logarithmic in $w = 32$, which is smaller than the maximal number of processors. This size was chosen based on empirical testing. The root node and its children contain two prisms of size 32 and 8 for the root and 16 and 4 its children. The nodes at depths 3, 4, and 5 have a single prism of size 2, 1, and 1, respectively. The `spin` is equal to 32, 16, 8, 4, and 2 for balancers at depths 0, 1, 2, 3, 4, and 5, respectively.

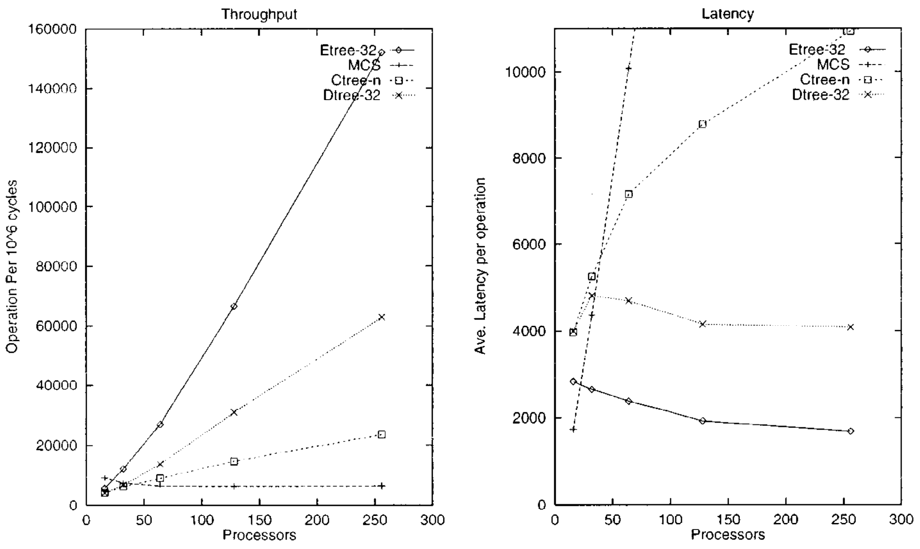


Fig. 7. Produce–consume: throughput and latency with `Workload= 0`.

From Figure 7 we learn that under high loads diffracting and elimination trees provide the most scalable high-load performance. However, as observed by Shavit and Zemach [25], as the level of concurrency increases, while the diffracting tree manages only to keep the average latency constant, the average latency in the elimination tree continues to *decrease* due to the increased numbers of successful eliminating collisions taking place on the top levels of the tree. The effect on the throughput is an up to 2.5 times increase in requests that are answered by the elimination tree! The percentage of eliminated tokens at the root varies between 44.7% when only 16 processors are participating and up to 49.7% for 256 processors. In fact, as can be seen from Table 1, most enqueue/dequeue requests never reach the lower level balancers, and the expected number of balancers traversed (including the pool at the leaf) for 16 processors is 3.14 nodes (38.9% of the requests access the leaf pools) and for 256 processors 2.082 nodes (only 8.95% of the requests eventually access the pools at the leaves). As seen in Figure 7, at such high levels of concurrency the elimination tree is almost as fast as the MCS-queue-lock is when there are just a few processes.

Table 1. Fraction of tokens eliminated per tree level.

	16 Processors	256 Processors
Level 0	44.7%	49.8%
Level 1	24%	49.1%
Level 2	5.8%	45.2%
Level 3	1.9%	32.9%
Level 4	0%	6.8%

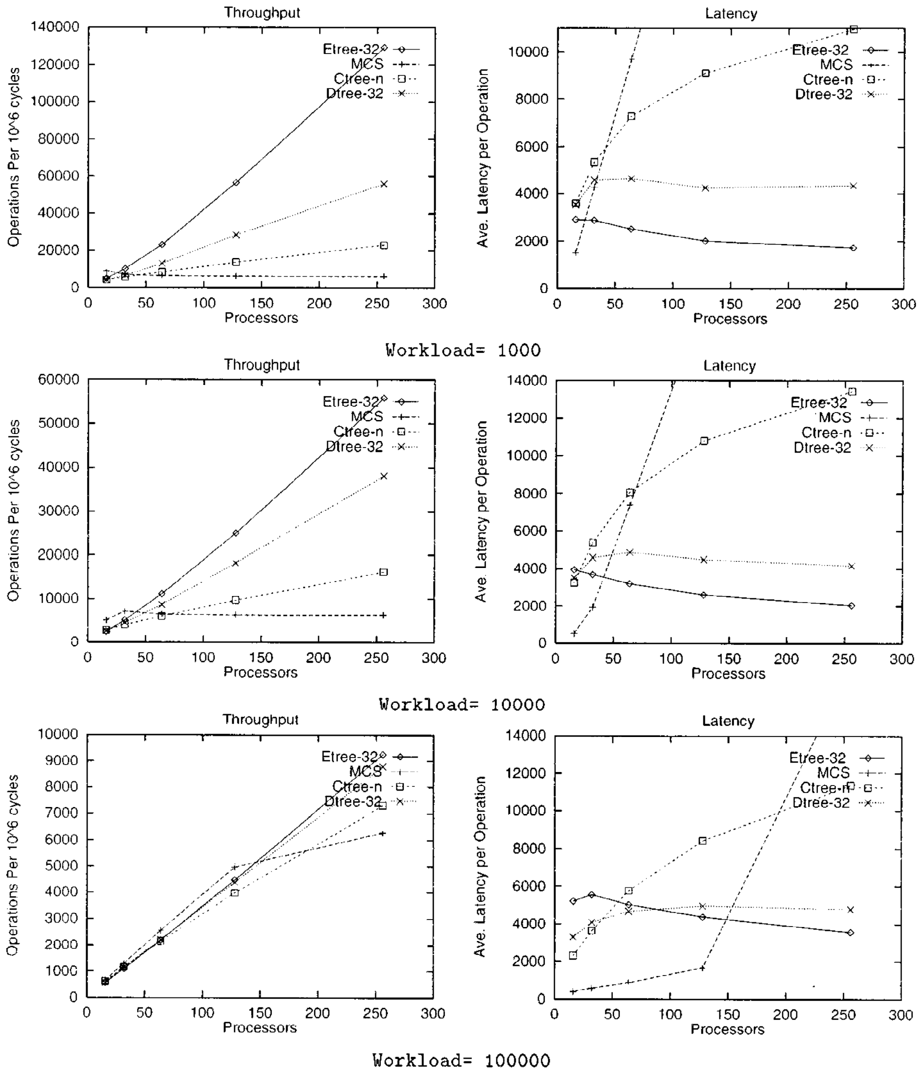
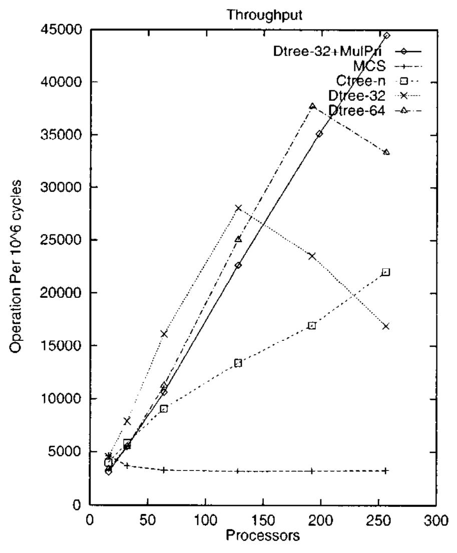


Fig. 8. Produce–consume: throughput and latency with `Workload > 0`.

In Figure 8 we compared the various methods as access patterns become more sparse. The MCS lock outperforms all others when the number of processes is small, and, unlike in the high-load case of Figure 8, even with a high number of processes the elimination tree cannot match its low latencies because of the low levels of elimination on the root balancer. As the chances of combining, diffraction, and elimination drop, the depth of the structures comes more into play. For 256 processors the optimal combining tree requires $2 \log n = 16$ node traversals (up and down the tree), while the optimal width 32 diffracting and elimination trees have depth 5 and thus require far fewer operations.



```
repeat
  fetch_and_inc();
until 10^6 cycles elapsed
```

Fig. 9. The Counting benchmark.

It follows that the elimination and diffracting tree performance graphs converge, and at sufficiently high levels of concurrency remain far better than the combining tree.

2.5.2. *The Counting Benchmark.* Our new multilayered prism approach is slightly more costly but scales better than the original single prism construction of Shavit and Zemach [25], since it increases the likelihood of successful collisions. This conforms with the steady-state modeling of diffracting trees by Shavit *et al.* [24]. As can be seen from Figure 9, when running a benchmark of *fetch&increment* operations where no *eliminating collisions* can occur, the DTREE[32] and DTREE[64] with original single `Prism` balancers outperform a DTREE[32] with our new multilayered balancers in almost all the levels of concurrency which could be incurred in the 256-processor produce–consume benchmark (on average each DTREE[32] has 128 or so concurrent enqueues). However, unlike our multilayered balancer constructions, they do not continue to scale well at higher levels of concurrency.

2.5.3. *The Response Time Benchmark.* We compared elimination trees with the randomized method of Rudolph *et al.* (RSU) [22], which we chose as a representative of the class of *load-balanced local pools* methods, which also include the randomized methods of Kotz and Ellis [15] (RSU is a refinement of this method), of Lüling and Monien [17] (this method is a refinement of RSU), and the job-stealing method of Blumofe and Leiserson [6]. We did not compare with Manber's deterministic method [19] as Kotz and Ellis [15] have shown empirically that the randomized methods tend to give better overall performance. It should be kept in mind that there are various situations in which any one of these techniques outperforms all the others and vice versa.

The RSU scheme is surprisingly simple:

RSU A processor enqueues tasks in its private task queue. Before dequeuing a task, every processor flips a coin and executes a *load-balancing* procedure with probability $1/l$, where l is the size of its private task queue. Load balancing is achieved by first choosing a random processor and then moving tasks from the longer task queue to the smaller to equalize their sizes.

We note that under high loads, and especially in applications such as job-distribution where each process performs both enqueues and dequeues, these methods are by far superior to elimination trees and all other presented methods. (The 10-queens benchmark in the left-hand side of Figures 10 and 11 is a lesser example of RSU's performance. Initially one processor generates 10 tasks of depth 1 simultaneously. Each one of n processors repeatedly dequeues a task and if the task's depth is smaller than 3 it waits $work = 8000$ cycles and enqueues 10 new tasks of depth increased by one.) However, as we know from theoretical analysis, their drawback is the rather poor $\Theta(n)$ expected latency when there are sparse access patterns by producers and consumers that are trying to pass information from one to the other, as could happen, say, in an application coordinating sensors and actuators.

The right-hand side of Figures 10 and 11 show the results of an experiment attempting to evaluate (in a synthetic setting of course) how much this actually hampers performance, by measuring the average latency incurred by a dequeue operation trying to find an element to return. We do so by running our 256-processor machine with $n/2$ processors as enqueuers and $n/2$ as dequeuers, where n varies between 2 and 256.

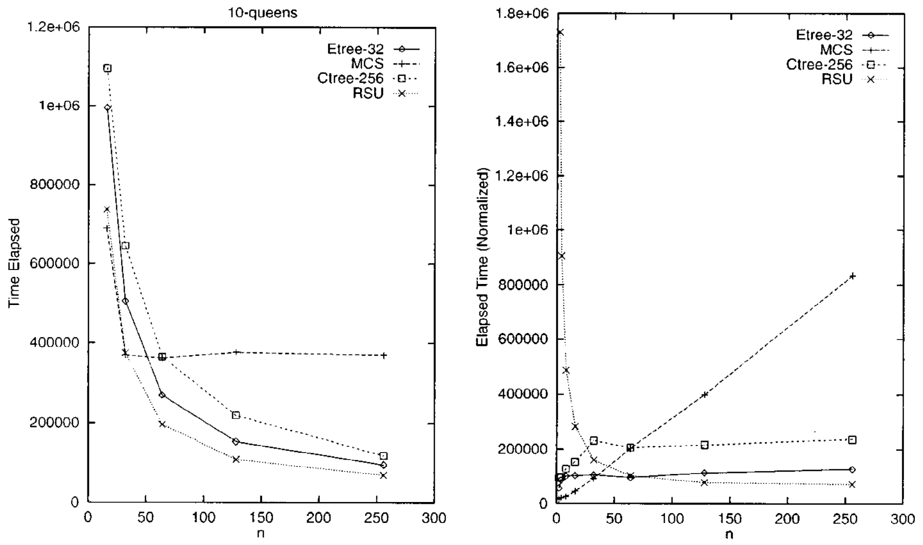


Fig. 10. 10-Queens and response time graphs.

```
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
```

```
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. 11. Code for the 10-Queens and response time benchmarks.

Each one of the enqueuing processors repeatedly enqueues an element in the pool and waits until the element has been dequeued by some dequeuing process. Each time we measured the time elapsed between the beginning of the benchmark until 2560 elements were dequeued, and normalized by the number of dequeue operations per process. Note that because of the way it is constructed, there is no real pipelining of enqueue operations, and this benchmark does not generate the high workload of the produce–consume benchmark for large numbers of participants.

As can be seen, RSU does indeed have a drawback since it is almost 100 times slower than the queue-lock and 30 times slower than an elimination tree for sparse access patterns. This is mostly due to the fact that the elimination tree even without eliminating collisions will direct tokens and antitokens to the same local piles within $O(\log w)$ steps. RSU reaches a crossover point when about a quarter of all local piles are being enqueued into. In summary, elimination trees seem to offer a reasonable middle-of-the-way response time over all ranges of concurrency.

3. Stack-Like Pools

Many applications in the literature that benefit by keeping elements in LIFO order would perform just as well if LIFO were kept among all but a small fraction of operations. LIFO-based scheduling will not only eliminate excessive task creation in many cases, but it will also prevent processors from attempting to dequeue and execute a task which depends on the results of other tasks [26]. Blumofe and Leiserson [6] provide a scheduler based on a randomized distributed pool having stack-like behavior on the level of local pools. We present here a construction of a pool that globally behaves like a stack. Our construction is based on the use of an elimination tree to create a single counter that can be both incremented and decremented concurrently, and can thus serve as a high bandwidth pointer to the head of the stack.

3.1. *Increment–Decrement Counting Trees*

We define a new type of balancer, the *gap elimination balancer*, that allows both tokens and antitokens as inputs, and balances the “difference” between them (the surplus of tokens over antitokens) on its output wires. We use *gap elimination balancer*s to construct counting trees that allow both increments and decrements. It has recently been shown by two independent teams, Busch and Mavronicolas [8] and Aiello *et al*. [3] that the increment/decrement properties we describe hold for counting networks in general, not only for trees.

A *gap elimination balancer* is a *elimination balancer* that in addition to the quiescence and pairing property must satisfy the additional requirement that:

Gap Step Property. In any quiescent state $0 \le (y_0 - \bar{y}_0) - (y_1 - \bar{y}_1) \le 1$.

In other words, any surplus of tokens over antitokens on the balancers output wires is distributed so that there is a gap of no more than one token on wire 0 relative to wire 1 in any quiescent state. Clearly, the gap step property implies the *pool-balancing* property on the balancer's output wires.

Claim 3.1. *Every gap elimination balancer satisfies the pool-balancing property.*

We design IncDecCounter$[w]$ as a *counting tree* [25] (a special case of the structure with regular token routing balancers replaced by token/antitoken routing *gap elimination balancers*). For w a power of two, IncDecCounter$[2k]$ is just a root gap balancer connecting to two IncDecCounter$[k]$ trees with the output wires $y_0, y_1, \ldots, y_{k-1}$ of the tree hanging from wire "0" redesignated as the even output wires $y_0, y_2, \ldots, y_{2k-2}$ of IncDecCounter$[2k]$, and the wires of the tree extending from the root's "1" output wire redesignated as the odd output wires $y_1, y_3, \ldots, y_{2k-1}$.

Lemma 3.2. *The* IncDecCounter$[w]$ *tree constructed from gap elimination balancers has the gap step property on its output wires, that is, in any quiescent state*

$$0 \le (y_i - \bar{y}_i) - (y_j - \bar{y}_j) \le 1$$

for any $i < j$.

Proof. We use that fact that the layout of the IncDecCounter is identical to that of a counting tree [25], in order to show that if for some execution the IncDecCounter reaches a quiescent state which does not satisfies the gap step property, then there is an execution of the counting tree in which the step property is violated too. This is a contradiction to Theorem 5.5 of [25]. Let T^g be an IncDecCounter constructed from gap balancers g, and let T^b be the isomorphic counting tree which is the result of replacing every gap balancer g in IncDecCounter by a regular balancer b. Given an execution history h^g of T^g, for every gap balancer g, let h_x^g be the gap between tokens and antitokens on g's input wire x, and let h_0^g and h_1^g be the gap at each of g's output wires y_0 and y_1. Define h_x^b, h_0^b, and h_1^b for h^b of T^b analogously.

Assume that for some execution history h^g of T^g, the gap step property is violated in a quiescent state. Assume first that the total difference between the number of tokens and antitoken accessing T^g is some nonnegative number G. Let h^b be an execution of T^b in which G tokens access the tree T^b. By a simple inductive argument used on the depth of the trees, it can be shown that for every gap balancer g in T^g and its matching balancer b in T^b, the following holds: $h_x^g = h_x^b \wedge h_0^g =_0^b \wedge h_1^g = h_1^b$. Consequently, it follows that:

Claim 3.3. *If for some execution history h^g of T^g, where G is nonnegative, the gap step property is violated in a quiescent state, then it is also violated for the matching history h^b of T^b.*

Assume now that, for h^g, the difference G between the total number of tokens and antitokens is negative. Let k be the smallest number such that $2^d * k + G \geq 0$ where d is the depth of the tree. Let $h1^g$ be an execution of T^g, in which after the completion of h^g, $2^d * k$ tokens were pushed through T^g. Using a simple inductive argument on the depth of the tree, it can be shown that, for every node g of depth d' in T^g, $h_x^g + k * 2^{d-d'} = h1_x^g$. Therefore, since k tokens will have been added equally to all the exits of T^g, the gap step property will be violated in $h1^g$ too. Since, in $h1^g$, the gap at the entrance of the tree is nonnegative, the claim follows by applying Claim 3.3. □

A *stack-like pool* is constructed, as with the pool data structure, by placing sequentially accessed "local stacks" at the leaves of an INCDECCOUNTER[w] tree. The following theorem is a corollary of Theorem 2.2 and Claim 3.1:

Theorem 3.4. *The stack-like pool construction is a correct pool implementation.*

The next theorem, which explicates the the LIFOish behavior of a stack-like pool is a direct corollary from the step property of Lemma 3.2, and is left to the interested reader.

Theorem 3.5. *In any sequential execution the stack-like pool provides a LIFO order on enqueues and dequeues.*

In Section 3.5 we present empirical evidence that suggests that even though the stack-like pool is not linearizable [14] to a sequential stack, it is linearizable in executions without severe timing anomalies, hence our use of the term "stack-like."

3.2. *Implementing the Gap Elimination Balancer*

The pool elimination balancer construction from the former section can be modified so that it satisfies the gap step property. This is done by replacing Part 2 of the code in Figure 4 with the following:

```
AquireLock(b $\rightarrow$ Lock);
if compare_and_swap(Location[mypid], $\langle$b,my_type$\rangle$, $\langle$0,EMPTY$\rangle$) then
  i:= b $\rightarrow$ INCDECtoggle;
  b $\rightarrow$ INCDECtoggle := Not(i);
  ReleaseLock(b $\rightarrow$ Tokens[mytype]);
  return b $\rightarrow$ OutputWire[i];
```

```
else
  ReleaseLock(b $\rightarrow$ Lock);
  if Location[mypid]= $\langle$0,DIFFRACTED$\rangle$ return (b $\rightarrow$ OutputWire[0])
  else return ELIMINATED
```

Instead of accessing two different toggle bits, both tokens and antitokens use the same toggle bit `INCDECtoggle`. If a token does not collide in the prisms, it toggles `INCDEC-toggle` and chooses an output wire according to the old value of the bit. An antitoken similarly toggles `INCDECtoggle`, but it chooses an output wire according to the *new* value of `INCDECtoggle` (using machine language notation, tokens perform a `fetch&complement` and antitokens a `complement&fetch`). On an intuitive level, this combination causes an antitoken to "trace" the last inserted token.

3.3. *Correctness Proof of Gap Balancer Implementation*

In order to prove the correctness of our gap balancer implementation we first show that all the tokens that have accessed the toggle bit satisfy the gap step property. As before, let t_i and $\bar{t}_i$ be the number of toggling tokens and antitokens exiting the balancer on wire i.

Lemma 3.6. *In any quiescent state* $0 \le (t_0 - \bar{t}_0) - (t_1 - \bar{t}_1) \le 1$.

Proof. The proof is by induction on the length of the history h of accesses to the toggle bit. If history h contains only token transitions or only antitoken transitions, then the property holds trivially. If h consists of transitions of both token types, there must be at least one token transition τ and one antitoken access $\bar{\tau}$ which followed one other in the history. We define h' to be the history h without τ and $\bar{\tau}$. Since following τ and $\bar{\tau}$ the `INCDECtoggle` bit returns to the same state it was before these transitions accessed it, h' is a possible history of the access to `INCDECtoggle` and by induction hypothesis satisfies the step property. Now, since both τ and $\bar{\tau}$ leave on the same output wire, h also satisfies the balancing property. □

Since the elimination protocols are identical in both the pool and gap elimination balancer implementations, the proof of the following three lemmas are identical to the proofs of Lemmas 2.3, 2.4, and 2.5, respectively, and are therefore omitted.

Lemma 3.7. *For every process p, if in a given state* `Location[`p`] =` $\langle b, * \rangle$, *then p is executing* `TokenTraverse` *on balancer b.*

Lemma 3.8. *Every token traversing a balancer b can be diffracted or eliminated by at most one other token.*

Lemma 3.9. *A toggling, eliminating, or diffracting token T_p cannot be eliminated or diffracted by some other token T_q.*

We can now conclude the correctness proof of our gap balancer implementation:

Theorem 3.10. *The gap eliminating balancer implementation satisfies the gap step property.*

Proof. Using the same notations as in the correctness proof of the pool balancer, we know from Lemmas 3.7, 3.8, and 3.9 that $\bar{e} = e$, $d_0 = d_1$, and $\bar{d}_0 = \bar{d}_1$. Therefore $(t_0 - \bar{t}_0) - (t_1 - \bar{t}_1) = ((t_0 + d_0) - (\bar{t}_0 + \bar{d}_0)) - ((t_1 + d_1) - \bar{t}_1 + \bar{d}_1)$. Since, $y_0 = t_0 + d_0$, $y_1 = t_1 + d_1$, $\bar{y}_0 = \bar{t}_0 + \bar{d}_0$, and $\bar{y}_1 = \bar{t}_1 + \bar{d}_1$ we may conclude that $0 \le (y_0 - \bar{y}_0) - (y_1 - \bar{y}_1) \le 1$. □

3.4. *Performance of the Stack-Like Pool*

We tested the performance of the stack-like pool for the produce–consume benchmark from Section 2. We implemented an INCDECCOUNTER[32] with prism sizes and spin times as in the POOL[32]. In Figure 12 we present the result of a comparision between an INCDECCOUNTER[32]-based stack-like pool and a POOL[32] in the producer-consumer benchmark under high load `Workload = 0`. As can be seen, though tokens are accessing a shared toggle bit instead of two separate ones, high elimination rates on the prisms allow the efficiency of the stack-like pool to fall only slightly from that of the POOL[32].

3.5. *Almost Linearizability*

Herlihy and Wing's linearizability [14] is a consistency condition that specifies the allowable concurrent behaviors of an object by way of a mapping to a sequentially specified object whose behaviors are easy to state. A linearization mapping exists if a point can be picked within the execution interval of every concurrent operation so that the collection of operations executed sequentially according to the order among these points, meets the sequential object specification. We present some empirical evidence that suggests that even though the stack-like pool is not always linearizable to a sequential stack, it behaves very much like one.

Given a stack-like pool implementation, let $E(e)$ and $D(e)$ respectively denote an enqueue operation of e and a dequeue operation returning e. Let $\rightarrow$ be the real-time order

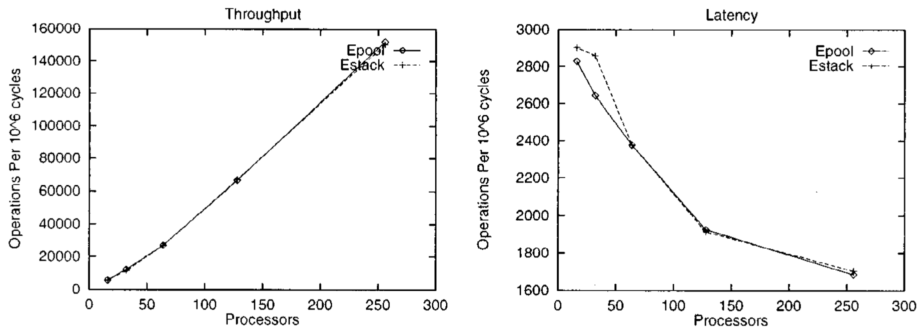


Fig. 12. Comparison between a pool and a stack-like pool.

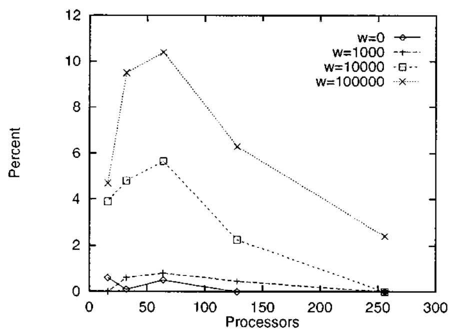


Fig. 13. Produce–consume: percentage of dequeue operations that are not linearizable.

between the operations ($OP_1 \to OP_2$ iff OP_1 has terminated before OP_2 has started). We say that the operation $D(x)$ in an execution e is *not linearizable* if there are $E(y)$, $E(x)$ such that $E(x) \to E(y) \to D(x)$ and either $D(y)$ does not exist in e or $D(y)$ exists in e and $E(x) \to E(y) \to D(x) \to D(y)$. A stack-like pool implementation is *linearizable* [14] if it ensures that every execution does not contain a dequeue operation that is not linearizable.

Our elimination-tree-based INCDECCOUNTER[w] is easily shown to be not linearizable to a sequential counter with increments and decrements. However, we present in Figure 13 empirical evidence suggesting that scenarios in which the linearizabilty of our stack-like pool is violated require extreme timing anomalies that it might be argued are not likely to occur frequently. We ran the producer-consumer benchmark where each processor, after traversing a balancer node, waits a random number of cycles between 0 and $W = 0$, 1000, 10,000, 100,000 until 2000 dequeue operations are executed. The graph presented plots the percentage of dequeue operations that are not linearizable. Note that for tightly synchronized executions ($W = 0$), our stack-like implementation is linearizable to a stack at almost all levels of concurrency.

4. Conclusions and Further Research

Our paper introduces the notion of "antitokens" to allow decrement operations on a counting-tree [25]. Since the initial publication of our results [2], two independent research teams, Busch and Mavronicolas [8] and Aiello *et al.* [3], have recently extended our proofs to show that counting networks [5] in general, not only trees, work with antitokens (Busch and Mavronicolas [8] show this also for multibalancers [2], [10], that is, balancers with multiple inputs and output wires).

In summary, *elimination trees* represent a new class of concurrent algorithms that we hope will prove an effective alternative to existing solutions for produce/consume coordination problems. This paper presents shared memory implementations of elimination trees, and uses them for constructing pools and stack-like pools.

There is clearly room for experimentation on real machines and networks. Given the hardware *fetch-and-complement* operation to be added to the Alewife machine's

Sparcle chip's set of colored load/store operations [16], a shared memory elimination-tree will be able to be implemented in a wait-free manner, that is, without any locks. Our plan is to test such "hardware supported" elimination-tree performance. We also plan to develop better measures and methods for setting the tree parameters such as prism `size` and balancer `spin`, and are currently developing message passing versions of our algorithms.

Acknowledgments

We would like to thank Yehuda Afek, Bill Aiello, Maurice Herlihy, and Asaph Zemach for their many helpful comments.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture & Performance. In *Proceedings of the* 22*nd International Symposium on Computer Architecture*, pages 2–13, Santa Margherita, Ligure, Italy, June 1995. Also, MIT/LCS Memo TM-454, 1991.

[2] E. Aharonson and H. Attiya. Counting Networks with Arbitrary Fan Out. *Distributed Computing*, 8(4):163–169, 1995. Also, Technical Report 679, The Technion, June 1991.

[3] W. Aiello, M. Herlihy, N. Shavit and D. Touitou. Inc/Dec Counting Networks. Manuscript, December 1995.

[4] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[5] J. Aspnes, M. P. Herlihy, and N. Shavit. Counting Networks. *Journal of the ACM*, 41(5):1020–1048, September 1994.

[6] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of the* 35*th Symposium on Foundations of Computer Science*, pages 365–368, November 1994.

[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-561, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1991.

[8] C. Busch and M. Mavronicolas. The Strength of Counting Networks. In *Proceedings of the* 15*th Annual ACM Symposium on Principles of Distributed Computing*, page 311, Philadelphia, PA, May 1996.

[9] D. Chaiken. Cache Coherence Protocols for Large-Scale Multiprocessors. S. M. thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Technical Report MIT/LCS/TR-489, September 1990.

[10] E. W. Felten, A. LaMarca, and R. Ladner. Building Counting Networks from Larger Balancers. T.R. #93-04-09, University of Washington.

[11] D. Gawlick. Processing "Hot Spots" in High Performance Systems. In *Proceedings COMPCON* '85, pages 249–251, San Francisco, CA (30th IEEE Comp. Society International Conference), February 1985.

[12] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proceedings of the* 3*rd ASPLOS*, pages 64–75, April 1989.

[13] M. Herlihy, B. H. Lim, and N. Shavit. Scalable Concurrent Counting. *ACM Transactions on Computer Systems*, 13(4):343–364, 1995. Full version available as a DEC TR.

[14] M. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transaction on Programming Languages and Systems*, 12(3):463–492, July 1991.

[15] D. Kotz and C. S. Ellis. Evaluation of Concurrent Pools. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 378–385, June 1989.

[16] J. Kubiatowicz. Personal communication (February 1995).

[17] R. Lüling and B. Monien. A Dynamic Distributed Load Balancing Algorithm with Provable Good Performance. In *Proceedings of the* 5*th ACM Symposium on Parallel Algorithms and Architectures*, pages 164–173, June 1993.

[18] N. A. Lynch and M. R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proceedings of the Sixth ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing*, August 1987, pp. 137–151. Full version available as Technical Report MIT/LCS/TR–387, Laboratory for Computer Science, Massachusetts Institute of Technology.

[19] U. Manber. On Maintaining Dynamic Information in a Concurrent Environment. *SIAM Journal on Computing*, 15(4):1130–1142, November 1986.

[20] 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, Feb. 1991.

[21] G. H. Pfister and A. Norton. "Hot Spot" Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, 34(11):933–938, November 1985.

[22] L. Rudolph, M. Slivkin, and E. Upfal. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. In *Proceedings of the* 3*rd ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, July 1991.

[23] N. Shavit and D. Touitou. Elimination Trees and the Construction of Pools and Stack. In *Proceedings of the* 7*th Annual Symposium on Parallel Algorithms and Architectures* (*SPAA*), pages 54–63, July 1995.

[24] N. Shavit, E. Upfal, and A. Zemach. A Steady-State Analysis of Diffracting Trees. In *Proceedings of the* 8*th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 33–41, Padua, Italy, June 1996. Also, to appear in special issue of *Theory of Computing Systems*.

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

[26] K. Taura, S. Matsuoka, and A. Yonezawa. An Efficient Implementation Scheme of Concurrent Object-Oriented Languages on Stock Multicomputers. In *Proceedings of the* 4*th Symposium on Principles and Practice of Parallel Programming*, pages 218–228, May 1993.

Received February 28, 1996, *and in final form January* 23, 1997.