Elimination Trees and the Construction of Pools and Stacks

(PRELIMINARY VERSION)

Nir Shavit* — MIT and Tel-Aviv University

Dan Touitou — Tel-Aviv University

April 26, 1995

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* and *stacks* are two structures that 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 [17]. A *pool* (also called a pile [13], 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 LIFO ordering.

The literature offers us a variety of possible pool implementations. On the one hand there are queue-lock based solutions such as of Anderson [1] and Mellor-Crummey and Scott [8], 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 wonderfully simple and effective randomized *work-pile* and *job-stealing* techniques of Rudolph, Slivkin-Allaluf and Upfal [6, 13], that 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). Diffracting trees [16] have been proposed as a reasonable middle-of-the-road solution. They guarantee termination within $O(\log w)$ time (where $w << n$) under sparse access patterns, and rather surprisingly manage to maintain similar average response time under heavy loads.

1.1 Our results

This paper introduces *elimination trees*, a novel form of diffracting trees that offers pool and stack implementations with the same $O(\log w)$ termination guarantee under sparse patterns, but with 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 the "probabilistic" methods [13], providing improved response time as the load on them increases.

In a manner similar to diffracting trees, elimination trees are 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 *anti-tokens*. 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 anti-tokens left and right in a balanced manner. For example, in the case of a stack implementation, 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 anti-tokens toggle the bit and go left or right according to its *new* value. Now, imagine that stack locations are placed at the leaves of the tree, and think of tokens as enqueue requests and anti-tokens as dequeue requests. Figure 1 shows a width four tree after 3 enqueues (tokens) and a dequeue (anti-token) 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

[1] Contact Author E-mail: shanir@theory.lcs.mit.edu

[1] They guarantee that a dequeue operation on a non-empty queue will always succeed

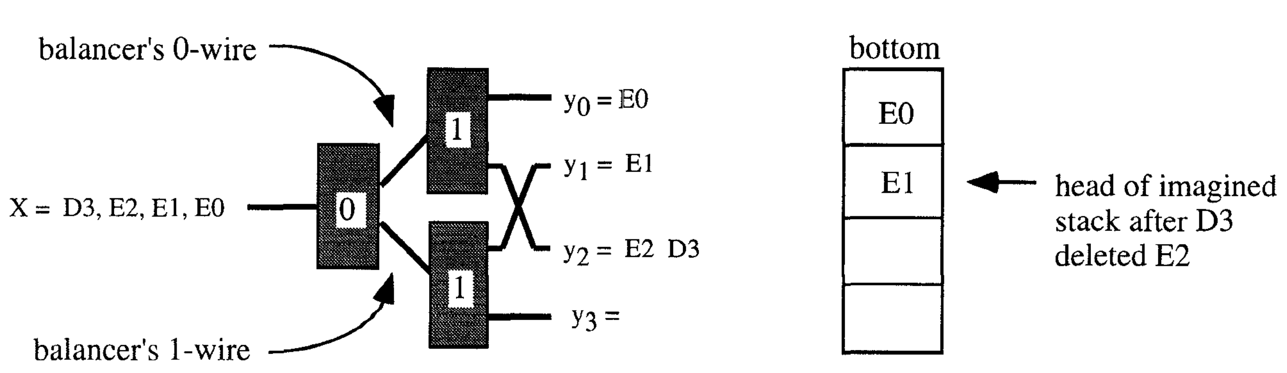


Figure 1: A sequential execution on a STACK[4] elimination tree

if next a token will enter it will see 0 and then 1 and end up on wire y_2, while if the next to enter is an anti-token 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 tree construction is a novel form of a *counting network* [3] based counter, that allows decrement (anti-token) operations in addition to standard increment (token) operations.

However, this simple approach is bound to fail since the toggle bit at root of the tree will be a hot-spot [9, 10] and a sequential bottleneck that is no better than a centralized stack implementation. The problem is overcome by placing a *diffracting prism* [16] structure in front of the toggle bit inside every balancer. 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 anti-tokens, not only tokens, are diffracted. The second, more important fact, is that it will continue to work if collisions among a token and an anti-token result in the "elimination" of the pair, without requiring them to continue traversing the tree! In other words, a token and anti-token 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 anti-tokens 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 to other known methods using the Proteus Parallel Hardware Simulator [7] in a shared memory architecture similar to the Alewife machine of Agarwal et al. [2]. We found that elimination trees scale *substantially* better than all methods known to perform well under sparse loads, including queue-locks, software combining trees [4] and diffracting trees. They are inferior to the probabilistic techniques of [6, 13] (though in many cases not by much), especially for job distribution applications where a typical processor is the dequeuer of its latest enqueue. However, our empirical evidence suggests that elimination trees provide up to a factor of 30 better response time than [13] 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, we believe *elimination trees* offer a new approach to produce/consume coordination problems. This paper presents shared memory implementations of elimination trees, and uses them for constructing pools and almost-LIFO stacks. We are currently developing message passing versions.

2 Pools

A *pool* (also called a pile [13], centralized "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. Assume for simplicity that all enqueued elements e are unique. A pool is a relaxation of a FIFO-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 non-empty, that is, if the number of enqueue operations ever completed is greater or equal to the number of dequeue operations ever started.

A *successful* operation is one that is guaranteed to return an answer within finite (in our construction, *bounded*) time. Note that the decentralized techniques of [13] and [6] implement a weaker "probabilistic" pool definition, where condition *P2* is replaced by a *randomized* guarantee that dequeue operations succeed.

2.0.1 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 [16]. Our formal model follows that of Aspnes, Herlihy, and Shavit [3] using the I/O-automata of Lynch and Tuttle [11]. An *elimination balancer* is a routing element with one input wire x and two output wires y_0 and y_1. *Tokens* and *anti-tokens* arrive on the balancer's input wire at arbitrary times, and are output on its output wires. Whenever a token meets an anti-token inside the balancer, the pair is eliminated and the tokens are never output. We slightly abuse our notation and denote by x^t and $x^{\bar{t}}$ the number of tokens and anti-tokens ever received, and by y_i^t and $y_i^{\bar{t}}$ the number of tokens and anti-tokens ever output on the ith output wire. The balancer must guarantee:

Quiescence Given a finite number of input tokens and anti-tokens, it will reach a *quiescent* state, that is, a state in which all the tokens and anti-tokens that were not paired off and eliminated exited on the output wires.

Pool Balancing In any quiescent state, if $x^t \geq x^{\bar{t}}$ then for every output wire i, $y_i^t \geq y_i^{\bar{t}}$.

Let Pool$[w]$ be a binary tree of elimination balancers with a root input wire x and w designated output wires: $y_0, y_1, .., y_{w-1}$, constructed inductively by connecting the outputs of an elimination balancer to two Pool$[w/2]$ trees. We extend the notion of quiescence and pool balancing to trees in the natural way, claiming that

Lemma 2.1 *The outputs* $y_0, .., y_{w-1}$ *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. Assume the claim for Pool$[w/2]$ and let us prove it for Pool$[w]$. If the number of tokens entering the root balancer of Pool$[w]$ is greater or equal to the number of anti-tokens, then, by definition this property is kept on the output wires of the root balancer, and by the induction hypothesis on the output wires of both Pool$[w/2]$ trees. ■

On a shared memory multiprocessor, one can implement an elimination 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 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 "anti-token" through the network. 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 spin-lock protected queue will do). A process performs an enqueue operation by shepherding a token "carrying" the value the down the tree. If the token reaches the output wire, the associated value is enqueued in the small pool connected to that wire. The dequeue operation is similarly implemented by carrying an anti-token through the network. If this anti-token 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 anti-token'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 elegant randomized constructions of [6, 12, 13] 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 both under high and under low loads. This is not the case for [12, 13, 6] which provides exceptionally good behavior at high loads but can guarantee only an an expected $\Omega(n)$ behavior under sparse access patterns.

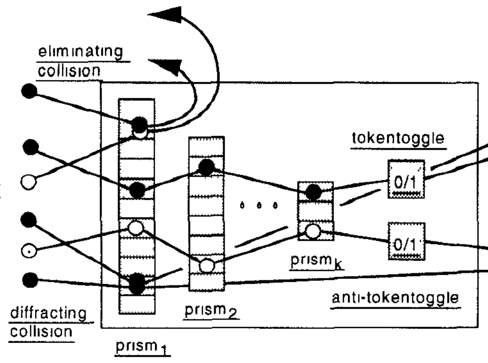
2.1 Elimination Balancers

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

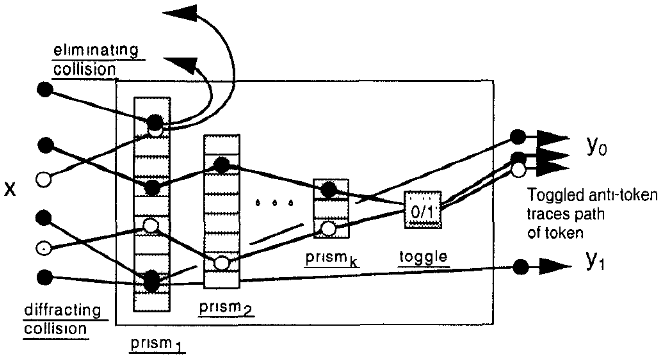
Diffracting balancers were introduced in [16]. Our shared memory construction of a diffracting elimination balancer, apart from providing a mechanism for token/anti-token elimination, also improves on the performance of the original diffracting balancer design. While a regular diffracting balancer [16] is constructed from a single prism array and a toggle bit, the elimination balancer we use in our pool construction (see lefthand side of Figure 2) has a sequence of prism arrays and two toggle bits, one for tokens and one for anti-tokens[2]. Each of the toggle bit locations is protected by an MCS-queue-lock [8]. A process shepherding a token or anti-token through the balancer decides on which wire to exit according to the value of the respective token or anti-token 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. anti-tokens) on the two output wires, so that there is in any quiescent state at most one token (resp. anti-token) more on the 0 output wire than on the 1 wire. The reader can easily convince herself that this suffices to guarantee the pool-balancing property. However, if many tokens were to attempt to access the same toggle bit concurrently, the bit would quickly become a hot spot. The solution presented in [16] 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 the sequel), 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 [16], 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 anti-tokens.

The key to our new constructions is the observation that for data structures which have complementary operations (such as enqueues and dequeues), one can can gain a substantial performance benefit from having a joined prism for

[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 anti-tokens



Pool collision balancer



Stack collision balancer

Figure 2: The structure of Pool and Stack elimination balancers

both tokens and anti-tokens. In addition to toggling and diffracting of tokens and anti-tokens, if a collision between a token and anti-token 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 anti-token toggle bits would have changed. Nevertheless, the combination of toggling, diffracting and elimination preserves the elimination balancer's correctness properties,[3] which by Lemma 2.1 guarantees pool-balancing. As can be seen from Table 1, at high levels of concurrency as many as half the tokens and anti-tokens can be eliminated on the first tree level alone, and only an eight are not toggled after passing the second level!

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, to many processes will collide on the same prism entry, creating a hot-spot. Unlike the single prism array of [16], 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. [4]

Figure 3 gives the code for traversing an elimination balancer (the code executed by an anti-token is almost symmetrical). Apart from reading and writing memory, it 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`.

[3]This is easy to see and prove for the pool type balancer which has two separate toggle bits, but harder for the single-bit stack balancers to be presented in the sequel

[4]We are currently testing reactive methods where prism width is varied dynamically in response to contention level

Our implementation also uses standard `AquireLock` and `ReleaseLock` procedures to enter and exit the MCS queue-lock [8].

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 $\mathtt{Prism}_1$ 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 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 it collision attempt (this eliminates race conditions). The second attempts to mark q's entry as "collided with p," notifying q of the collision type: `BY_TOKEN` or `BY_ANTI_TOKEN`. If both compare-and-swap operations succeed, the collision is successful, and p decides based on collision type to either 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 the 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 empty, and having failed to "collide with" another processor, spins on `location[`p`]` waiting for another processor to "collide with it." If during `spin` times no collision occurs, p restarts the whole process at the next level $\mathtt{Prism}_2$ 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. In case of an *eliminating collision*, token and anti-token can exchange values in the following way. Tokens write the value they carry in their `Location` entry instead of just writing `TOKEN`. When it has to eliminate an anti-token, a token writes its value in

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

Function TokenTraverse(b: ptr to balancer) returns (ptr to balancer or ELIMINATED);
   Location[mypid] := (b,TOKEN);
   /* 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) and (his_type = TOKEN or his_type= ANTI-TOKEN) then
              if compare_and_swap(Location[mypid],(b,TOKEN), EMPTY) then
                 if compare_and_swap(Location[him],(b,his_type), BY_TOKEN) then
                      return (b->OutputWire[1] or ELIMINATED if his_type = ANTI-TOKEN)
                  else
                      Location[mypid] := (b,TOKEN);
               else
                  return (b->OutputWire[0] or ELIMINATED if Location[mypid] = BY_ANTI_TOKEN)

        repeat b->Spin times /* wait in hope of being collided with */
            if  Location[mypid] = BY_TOKEN then return b->OutputWire[0];
            if  Location[mypid] = BY_ANTI_TOKEN then return ELIMINATED;

   /* Part 2  access toggle the bit */

   AquireLock(b->TokenLock);
   if compare_and_swap(Location[mypid],(b,TOKEN), EMPTY) then
         i:= b->TokenToggle;
         b->TokenToggle := Not(i);
         ReleaseLock(b->TokenLock);
         return b->OutputWire[i];
    else
         ReleaseLock(b->TokenLock);
         return (b->OutputWire[0] or ELIMINATED if Location[mypid] = BY_ANTI_TOKEN)
```

Figure 3: Traversing an eliminating balancer- the token's code

the anti-token's `Location` entry instead of `BY_TOKEN`. Finally, when it eliminates a token, an anti-token gets the value from the token's `Location` entry. In the full paper we prove by induction on the length of the computation that the above code implements an *elimination balancer*.

2.2 Performance

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 256 processors distributed-shared-memory machine similar to the MIT *Alewife* machine [2] of Agarwal et. al. Our simulations were performed using *Proteus* a multiprocessor simulator developed by Brewer, Dellarocas, Colbrook and Weihl [7]. Our preliminary results include several synthetic benchmarks.

2.2.1 Produce-Consume benchmark

We begin by comparing under high loads various *deterministic* pool constructions which are known to guarantee good enqueue /dequeue time when the load is low (sparse access patterns). In the produce-consume benchmark each processor alternates enqueueing a new element in the pool and dequeuing a value from the pool. We ran this benchmark varying the number of processors participating in the simulation during 10^6 cycles, measuring: *latency*, the average amount of time spent per produce and consume operation, 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 4).

We thus realized the centralized pool given in Figure 4, when the `NQcounter` and `DQcounter` are implemented using two counters of the following type:

MCS The MCS lock of [8], 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 *Fetch&Inc* using an optimal depth combining tree, whose response time is logarithmic in the maximal number of processors. An implementation of the software

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

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

Figure 4: A pool based on a shared counting.

combining tree protocol of Goodman et al. [4], modified according to [5]. Optimal width means that when n processors participate in the simulation, a tree of width $n/2$ will be used.

DTree A Diffracting Tree of width 32, using the optimized parameters of [16], whose response time is logarithmic in $w = 32$ which is smaller than the maximal number of processors.

and compare it to :

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. The root node and its children contain two prisms of size 32 and 8 for the root and 16 and 4 its the children. All other nodes contain only a single prism of size 2. The **spin** is equal to 32,16,8,4 and 2 for balancers at depths 0,1,2,3,4 and 5 respectively.

From Figure 5 we learn that diffracting and elimination trees provide the most scalable high load performance. However, as the level of concurrency increases, the diffracting tree manages only to keep the average latency constant, while the average latency in the elimination tree *decreases* due to the increased numbers of successful eliminating collisions taking place on the top levels of tree. The effect on the throughput is impressive: up to 2.5 times more requests are answered by the elimination tree! The fraction 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 request access the leaf pools) and for 256 processors 2.082 (only 8.95% of the request eventually access the pools at the leaves).

2.2.2 Counting Benchmark

Our new multi-layered prism approach is slightly more costly but scales better than the original single prism construction of [16]. As can be seen from Figure 6, 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 multi-layered balancers in almost all the levels

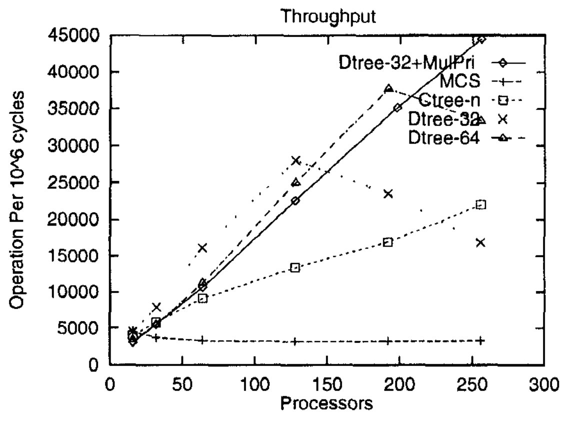


Figure 6: Counting benchmark

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 the multi-layered balancer constructions, they do not continue to scale well at higher levels of concurrency.

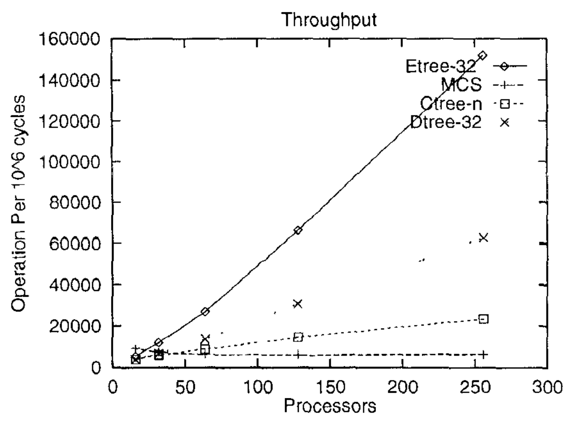
2.2.3 Response time benchmark

The randomized workpiles method of Rudolph, Silvkin-Allalouf and Upfal (RSU) [13] and later improvements by Lüling and Monien [12] are surprisingly simple:

RSU Processors enqueue task in a 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 bal-

	16 procs	256 procs
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%

Table 1: Fraction of tokens eliminated per tree level



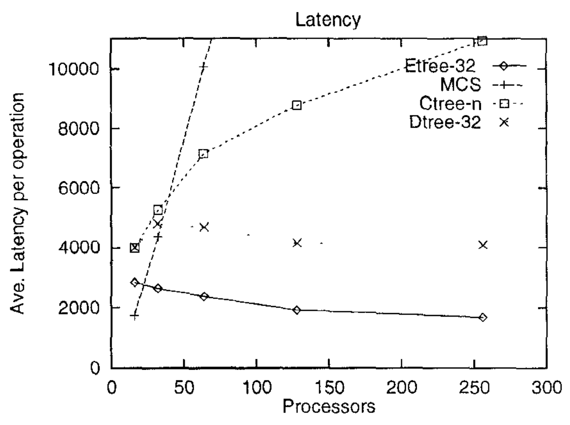


Figure 5: Produce-Consume: Throughput and Latency graphs

ancing is achieved by first choosing a random processor and then moving tasks from the longer task queue to the smaller so as 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 of Figure 7 is a lesser example of RSU's performance. [5]) 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 righthand side of Figure 7 shows 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 a 256 processor machine in which $n/2$ processors are enqueuers and $n/2$ are dequeuers where n varies between 2 and 256. 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 work-load 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 anti-tokens 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.

[5] Initially one processor, generates 10 tasks of depth 1 simultaneously. Each one of n processor repeatedly dequeues a task and if the task's depth is smaller than 3 it waits $work$ = 8000 cycles and enqueue 10 new tasks of depth increased by one

3 Almost Stacks

Many applications in the literature that benefit by keeping elements in *LIFO* order would perform just as well if *LIFO* would be kept among all but a small fraction of operations. LIFO-based scheduling will not only eliminate in many cases excessive task creation, but it will also prevent processors from attempting to dequeue and execute a task which depends on the results of other tasks [17]. Blumofe and Leiserson [6] provide a scheduler based on a version of the RSU algorithm having LIFO-ish behavior on a local level. We present here a construction of an *almost stack*. The full paper, following Herlihy and Wing's well accepted notion of linearizability for specifying concurrent data structures, provides a formal definition of *ϵ-linearizablity*, a variant of linearizablity that captures the notion of "almostness" by allowing a certain fraction of concurrent operations to be out-of-order. [6] An *almost stack* implementation is one whose operations are ϵ-linearizable to some sequential LIFO ordering.

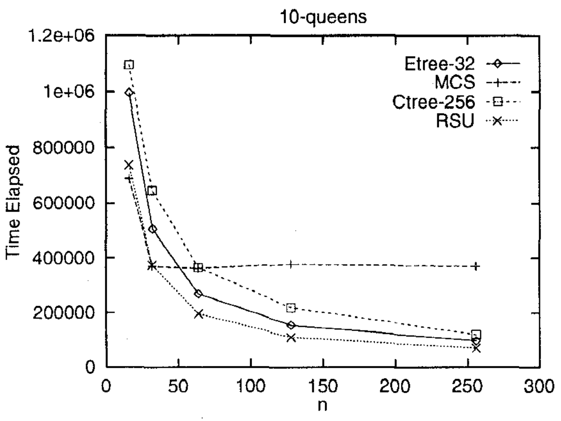
3.1 Elimination Stacks

A *stack elimination balancer* is a *pool elimination balancer* with the additional requirement that:

Gap Step Property In any quiescent state $0 \leq (y_0^t - y_o^{\bar{t}}) - (y_1^t - y_1^{\bar{t}}) \leq 1$.

In other words, any surplus of tokens over anti-tokens 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.

[6] This notion is remotely related to ϵ-serializablity [14], which allows individual database read transactions to return partially inconsistent states



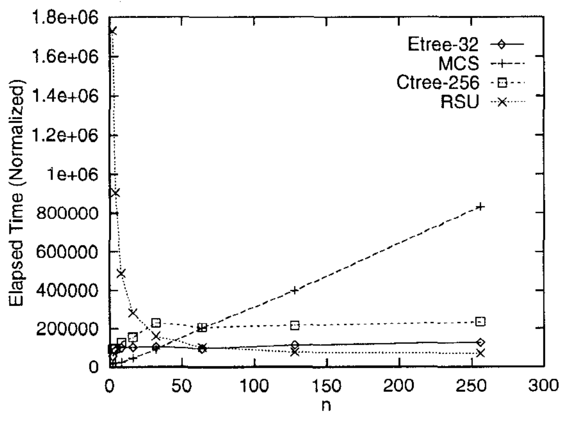


Figure 7: Comparison between RSU and Elimination pool

We design STACK[w] as a *counting tree* [16] (a special case of the structure with regular token routing balancers replaced by token/anti-token routing *Stack elimination balancers.* For w a power of two, STACK[$2k$] is just a root balancer connecting to two STACK[k] trees with the output wires $y_0, y_1, \ldots, y_{k-1}$ of the tree hanging from wire "0" re-designated as the even output wires $y_0, y_2, \ldots, y_{2k-2}$ of STACK[$2k$], and the wires of the tree extending from the root's "1" output wire re-designated as the odd output wires $y_1, y_3, \ldots, y_{2k-1}$.

Lemma 3.1 *A* STACK[w] *tree constructed from stack elimination balancers has the gap step property on its output wires, that is, in any quiescent state:*

$$0 \leq (y_i^t - y_i^{\bar{t}}) - (y_j^t - y_j^{\bar{t}}) \leq 1$$

for any $i < j$.

In fact, the STACK[w] tree is a novel form of a counting tree/network [3, 16], that allows both increment (token) and decrement (anti-token) operations.

Proof: Follows from the step property for counting trees (Theorem 5.5 of [16]) by replacing the step property (on tokens) for regular balancers by the gap step property (on token/anti-token difference) for stack elimination balancers. ■

An *almost-stack*is constructed as with the pool data structure by placing sequential "local stacks" at the leaves of a STACK[w] tree. It follows from Lemma 3.1 (and the reader is encouraged to try it out) that:

Corollary 3.2 *In any sequential execution the* STACK[w] *based construction is a LIFO stack.*

3.2 Eliminating Stack Balancer

One can modify the pool elimination balancer construction from the former section so that it satisfies the gap step property. Instead of accessing two different toggle bits, both tokens and anti-tokens use the same toggle bit `NQDQtoggle`. If a token does not collide in the prisms, it toggles `NQDQtoggle` and chooses an output wire according to the old value of the bit. An anti-token also toggles `NQDQtoggle`, but it chooses an output wire according to the *new* value of `NQDQtoggle`. In this way, the anti-tokens behave as if they "trace" the last inserted token.

Theorem 3.3 *The eliminating stack balancer satisfies the gap step property.*

Proof: (Sketch) Assume first that no collision occurs in the balancer. We prove that the that every history of the accesses to `NQDQtoggle` satisfies the property. We do so by induction on the length of the computation history. If history h contains only token transitions or only anti-token transitions then the property holds trivially. If h consists of transitions of both token types, there must be at least one token transition t and one anti-token access at which followed one other in the history. Let us define h' to be the history h without t and at. Since following t and at the `NQDQtoggle` bit remains in the same state it was before these transitions accessed it, h' is possible history of the access to `NQDQtoggle` and by induction hypothesis satisfies the step property. Now, since both t and at have been directed to the same output wire, h also satisfies the balancing property. Finally, since colliding tokens either disappear or are distributed equally between the output wires, the step property is satisfied. ■

The performance of the almost-stack is similar to that of the pool and is not presented for lack of space.

Given a 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 between the operations ($OP_1 \rightarrow OP_2$ iff OP_1 has terminated before OP_2 has started). We say that $D(x)$ in history h is *unlinearizable* if there are $E(y)$, $E(x)$ such that $E(x) \rightarrow E(y) \rightarrow D(x)$ and either $D(y)$ does not exist in h or $D(y)$ exists in h and $E(x) \rightarrow E(y) \rightarrow D(x) \rightarrow D(y)$. We empirically tested the ϵ-linearizability of our almost-stack implementation.

We ran the producer-consumer benchmark when each processor after traversing a node, waits a random number of cycles between 0 and $W = 0, 1000, 10000, 100000$ until 2000

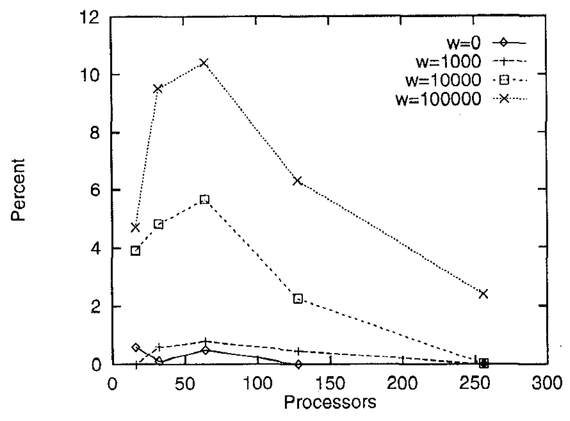


Figure 8: Produce-Consume: Percentage of unlinearizable Dequeue operations

dequeue operations were executed. The data presented is the fraction of unlinearizable dequeue operations, ones that return values that are inconsistent with a linearizable LIFO ordering of events. The results are in given in Figure 8. The y-axis shows the percentage of unlinearizable dequeue operations. Note that for tightly synchronized executions ($w = 0$), our implementation is linearizable to that of a stack at almost all levels of concurrency.

4 Conclusions and Further Research

Elimination trees represent a new class of concurrent algorithms that we hope will prove an effective alternative to the concurrent pool and stack algorithms in the literature.

There is clearly room for experimentation on real machines and networks. Given the hardware *fetch-and-complement* operation to be added to the Sparcle chip's set of colored load/store operations, one will be able to implement a shared memory elimination-tree 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 or methods for setting the tree parameters such as prism size and balancer spin.

References

[1] 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.

[2] A. Agarwal et al. 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, and appears as MIT/LCS Memo TM-454, 1991.

[3] J. Aspnes, M.P. Herlihy, and N. Shavit. Counting Networks. *Journal of the ACM*, Vol. 41, No. 5 (September 1994), pp. 1020-1048.

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

[5] M. Herlihy, B.H. Lim and N. Shavit. Low Contention Load Balancing on Large Scale Multiprocessors. *Proceedings of the 3rd Annual ASM Symposium on Parallel Algorithms and Architectures*, July 1992, San Diego, CA. Full version available as a DEC TR.

[6] R.D. Blumofe, and C.E. Leiserson. Sheduling Multithreaded Computations by Work Stealing. In *Proceeding of the 35th 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. MIT Technical Report /MIT/LCS/TR-561, September 1991.

[8] J.M. Mellor-Crummey and M.L. Scott Synchronization without Contention. In *Proceedings of the 4th International Conference on Architecture Support for Programming Languages and Operating Systems,* April 1991.

[9] G.H. Pfister and A. Norton. 'Hot Spot' contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(11):933–938, November 1985.

[10] D. Gawlick. Processing 'hot spots' in high performance systems. In *Proceedings COMPCON'85*, 1985.

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

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

[13] 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.

[14] Krithi Ramamrithan and Calton Pu. A Formal Characterization of Epsilon Serializability. *IEEE Transactions on Knowledge and Data Engineering*, 1994, to appear.

[15] M. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. In *ACM Transaction on Programming Languages and Systems,* 12(3), pages 463-492, July 199

[16] N. Shavit and A. Zemach. Diffracting Trees. In *Proceedings of the Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1994.

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