Combining Funnels

A new twist on an old tale...

Nir Shavit* Asaph Zemach†
Department of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel

Abstract

We enhance the well established software combining synchronization technique of Goodman et al. to create *combining funnels*. Previous software combining methods used a statically assigned tree whose depth was logarithmic in the total number of processors in the system. The new method allows to dynamically build combining trees with depth logarithmic in the actual number of processors accessing the data structure concurrently. The structure is comprised from a series of *combining layers* through which processor's requests are funneled. These layers use randomization instead of a rigid tree structure to allow processors to find partners for combining. By using an adaptive scheme the funnel can change width and depth to accommodate different access frequencies without requiring global agreement as to its size. Rather, processors choose parameters of the protocol privately, making this scheme very simple to implement and tune. When we add an "elimination" mechanism to the funnel structure, the randomly constructed "tree" is transformed into a "forest" of disjoint (and on average shallower) trees, thus enhancing the level of parallelism and decreasing latency.

We present two new linearizable combining funnel based data structures: a fetch-and-add object and a stack. We study the performance of these structures by benchmarking them against the most efficient software implementations of fetch-and-add and stacks known to date, combining trees and elimination trees, on a simulated shared memory multiprocessor using Proteus. Our empirical data shows that combining funnel based fetch-and-add always outperforms combining trees and the performance margin increases considerably if the number of processors is unknown in advance or if load is less than maximal. Elimination trees, which are not linearizable, prove to be 10% faster than funnels under highest load, but as load drops combining funnels adapt their size, giving them a 34% lead in latency.

*Contact Author: E-mail: shanir@math.tau.ac.il.
†Supported by an Israeli Ministry of Science Eshkol Scholarship.

PODC 98 Puerto Vallarta Mexico

1 Introduction

When different threads running a parallel application access the same object simultaneously, a synchronization protocol must be used to avoid interference. Since modern architectures usually supply very basic synchronization primitives, it is up to the programmer to handle more complex situations in software. Synchronization methods should be simple and easy to implement and offer both correctness and efficiency. Correctness implies that for any interleaving of instructions by any number of processors, the behavior of the synchronized object always adheres to some well defined specification. Efficiency, in this context, can be broken into several categories: parallelism – as more threads (processors) are added to the system the throughput should generally increase; scalability – it should be possible for the method to support an arbitrary number of threads; and robustness – the time it takes to perform operations should minimize sensitivity to load fluctuations. Finally, the method should be widely applicable to avoid the need to invent a new synchronization protocol for every application.

It is well documented [1, 7, 12, 21] that concurrent access to a single object by many threads can lead to a degradation in performance due to contention. A relatively well established method which has been used to alleviate this "hot spot" contention is *combining*. Originally designed by Gottlieb et al. to be used in switches of a network which connects processors to memory [9, 20], combining seeks to avoid contention by merging several messages with a like destination. If a switch discovers two read operations attempting to access the same word of memory, it will forward only one message to the memory system. When a message returns with the contents of the memory, the switch will dispatch two messages back to the processors to satisfy both read requests. In the NYU Ultracomputer [10], hardware switches can perform combining on several different kinds of messages, including reads, writes and fetch-and-add operations [15]. The most notable example of software combining for performing fetch-and-add are the combining trees of Goodman et al. [8] and Yew et al. [24]. In these algorithms, the current value of a fetch-and-add counter is stored at the root of a binary tree. Processors advance from the tree's leaves to its root, combining requests at each node along their path. Whenever combining occurs, one processor continues to ascend the tree, while the other is delayed. When a processor reaches the root, it adds to the counter the sum of all the fetch-and-add operations with which it combined, then it descends the tree, delivering the results to the delayed processors.

Combining trees are widely applicable and can be used to enhance the implementation of any fetch-and-Φ [10, 15] synchronization primitive, as well as some simple data structures, such as sets. Scalability as the number of processors P increases is achieved my making the tree deeper, adding more levels to make sure that the number of leaves is $\lceil P/2 \rceil$. Under maximal load, the throughput of such a tree will be $P/(2 \log P)$ operations per time unit, offering a significant speedup. Two mechanisms are used to keep contention at tree nodes low. Processors are statically pre-assigned two to a leaf, and every node contains a lock, processors must acquire this lock in order to ascend from a node to its parent. Thus, the number of processors that may concurrently enter a node is at most two, regardless of the load. In practice this means tree nodes can be made simple and require few instructions to traverse.

Combining is thus a compelling idea for providing linearizable parallel implementations. However, it turns out that the very mechanisms that make the tree so useful under high loads, namely static assignment and locking of nodes, are actually drawbacks as the load drops.

The downside of static assignment is that even if the tree is rarely accessed by all P processors simultaneously, its depth must still be $\log P$. The locking of tree nodes means that a processor that misses a chance for combining is locked out of the path to the root and must wait for an earlier one to ascend the tree and return before it can progress. As noted by Herlihy et al. [12], this makes combining trees extremely sensitive to changes in the arrival rate of requests. Herlihy et al. show that even a small drop from the maximal load will cause a 50% drop in the level of combining, and from there performance continues to degrade rapidly (this is discussed in detail in Section 4 of this paper and in [23]).

There is no obvious way of getting around these difficulties. For example, combining trees are not amenable to an adaptive strategy which might shrink the tree when average load is low (e.g. the reactive locks of Lim and Agarwal [16]) since there is no easy way to lower the number of nodes and still limit simultaneous access to a node to no more than two processors. Furthermore, decentralized algorithms for dynamically changing tree size (see for example the Reactive Diffracting Trees of Della-Libera and Shavit [6]) tend to be complex and require significant tuning efforts. Naive attempts to remove the locks and allow processors to pipeline requests up the tree would mean nodes (especially the root) could be reached by many processors at a time. The need to handle this increased parallelism and contention would complicate the protocol used in the tree nodes and increase latency.

In summary, is seems that allowing pipelining of requests, eliminating unnecessary waiting, and modifying the tree algorithm to be adaptive, would benefit performance. However, there seems to be no simple way to fit them into the binary combining tree framework.

1.1 Beyond trees: the new approach

In this paper we present a new general method for implementing the "combining" paradigm on shared memory multiprocessors. It allows us to reap most of the benefits of combining without the drawbacks of using a tree structure. Our method, *combining funnels*, replaces the "static" tree with a series of "randomized" *combining layers* through which requests are funneled and combined.

In broad terms, the combining funnel approach can be stated as follows. Given a simple data object with *combinable* [9, 15] operations, which operates correctly in a parallel environment (though not necessarily efficiently), parallel performance can be enhanced by adding a combining funnel structure as a front end. All processors attempting to access the object pass through the layers of the funnel and can collide with others heading for the same object. When a collision occurs the colliding processors perform a localized combining protocol much in the same way as communication network switches combine messages. When processors emerge from the funnel, they apply their (possibly combined) operation to the object.

The implementation of the layers of the funnel is based on a technique similar to the one used by the authors in the design of the diffracting tree *prism array* [23]. The prism array is a randomized software "bulletin-board" which allows pairs of processors to find each other and collide. While diffracting trees use these collisions to "diffract" pairs of processors, here we will use them to combine sets of requests. The *combining layer* is an array of locations $1..W$. A processor chooses an element uniformly at random from some subrange of $1..W$ and tries to combine with other processors who concurrently choose the same random element. The randomized access removes the need for static assignment of processors to elements, and since prism locations are never locked, unnecessary spin-waiting can be eliminated. The funnel itself is constructed from a cascade of combining layers leading to the central copy of the data structure. Each processor can independently and dynamically choose the depth (what layer) to start at and the subrange (what width) to randomly choose from at a given depth (see Figure 3 for a graphic depiction of this property). The choice is easily modified in response to the processor's local indicators of system load. The resulting protocol "adapts" the effective size of the data structure to the request load in a highly decentralized manner through convergence of the independent choices made by individual processors.

Adaptive algorithms, allowing the data structure to change behavior to accommodate different access frequencies, have been used both in locking (see Karlin et al. [14] and Lim and Agarwal [17]) and for more general fetch-and-Φ operations [16]. The work of Lim and Agarwal [16] showed the performance benefit of dynamically switching between locking an object and using (static) combining trees, based on whether the overhead of the latter justifies the added potential for parallelism. Combining funnels take this idea one step further by using a single funnel structure to support an entire range of sizes, from a simple lock to full funnel. Unlike the approach of Lim and Agarwal we do not require global agreement as to the size of the data structure, allowing each processor to choose the parameters of the funnel individually. This lack of coordination lowers overhead and simplifies the protocol. Though it means that different processors might end up with different size decisions: some using large funnels, some small, this does not affect the algorithm's correctness, and as we will show, produces significant performance advantage.

In a typical execution of the funnel, as operations collide many small combining trees are created, further collisions create larger trees and one ends up with a collection of concurrent combining trees "growing" towards the central structure. Trees are created and destroyed "on-the-fly" based on those processors concurrently passing through the funnel. Unless trees collide and combine, each tree advances independently. Since layers are never locked, slow trees don't delay faster ones, improving both parallelism and robustness. These and other combining funnel behaviors are shown in

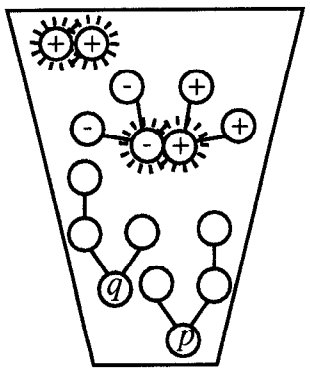

Figure 1: A whole slew of behaviors of a linearizable combining funnel based stack. Pushes are denoted by a "+" sign, and pops by a "−". Two pushing processors collide to create a new combining tree; a pair of opposite signed trees collide and, through local agreement, exit the funnel immediately; the tree rooted at q overtakes the tree rooted at p which is delayed for some reason.

Figure 1.

One can further enhance the performance of combining funnels for data structures that allow *elimination* in addition to combining (as defined in the work of Shavit and Touitou [22]). For example, we show that if the implemented data structure is a stack, one can preserve linearizability even if collections of "push" and "pop" operations that meet in a given layer of the funnel are eliminated. That is, the pushes return success and each pop returns one of the enqueued values it collided with, and all exit the funnel without ever reaching the central structure. In effect the tree structure is replaced by a forest of independent trees of requests, each of which can be satisfied in parallel.

This paper presents implementations of two combining funnel based data structures: a fetch-and-add object which can serve as a template for a general fetch-and-Φ object; and a concurrent stack. Both data structures are linearizable, a property found in combining trees but not in previous prism based methods such as diffracting trees and elimination trees. We studied their performance by benchmarking them against the most efficient software implementations known to date: combining trees and elimination trees. We found that the combining funnel based fetch-and-add performs considerably better than combining trees of fixed height. In a system of 256 processors where only 64 attempt to access the object, the fixed size tree is 8 levels deep (where 6 would suffice) and is outperformed by combining funnels by 70%. In stacks, under highest load, elimination trees provide 10% better latency. As the number of processors drops or local work between accesses increases, the trend is reversed and combining funnels gain the lead, e.g 34% lower latency at 32 processors. Unlike elimination trees whose latency increases as load on the object drops, the adaptive combining funnel has the property that decreasing load leads to an improvement in latency. In summary, our linearizable fetch-and-add and linearizable stack are by far the fastest such structures known to date.

The rest of the article is organized as follows. Section 2 presents the combining funnel scheme and gives an in-depth look at fetch-and-add and stacks, Section 3 describes how adaption is incorporated into the funnel, Section 4 gives benchmark results, and Section 5 concludes the paper and discusses areas of further research. In the appendix one can find detailed pseudo-code for both data structures implemented.

2 Combining Funnels

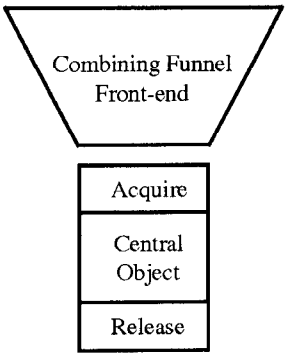

Figure 2: Schematic depiction of combining funnel mechanism

We first present our combining-funnel scheme in a generalized form and then show how both fetch-and-add and stacks fit into the framework. The idea, illustrated in Figure 2, is to maintain a single "central object" and use a series of funnel layers as a "front-end" to make access to it more efficient. Our only requirement in terms of parallelism from the central object is that it must correctly handle simultaneous access attempts by multiple processors. This can be achieved simply by protecting access to the object by locks. The combining funnel handles efficiency and prevents the object from becoming a serial bottle-neck.

Normally a processor would first acquire the object's lock, then apply its operation and finally release the lock. Instead, it will now first "pass through" a series of combining layers. The function of the layers is to hand each passing processor the ID of another processor that has recently gone through the same layer. Since each object has its own funnel this ID is likely to belong to a processor that is concurrently trying to access the same object. The first processor now attempts to *collide* with the one whose ID it got. If successful, processors can exchange information and update their operations accordingly. For example, processors p and q access a stack object concurrently with operations `PUSH(A)` and `PUSH(B)` respectively. Processor p passes through one of the stack's layers and exits with q's ID. If p manages to collide with q the results could be for p's operation to become `PUSH({A,B})` and q's to change to "`WAIT for B to be PUSHed`". We say p becomes q's parent since p is going to be performing both operations.

In a shared memory environment, a funnel layer can be implemented using an array. A processor arriving at the array picks a location at random and applies a register-to-memory-swap operation on it,[1] reading the ID written there and writing its own ID in its place. By overwriting existing IDs, we can keep the array up-to-date and avoid accumulating stale information. By using an array with several locations we allow many processors to pass through the layer at

[1] A read followed immediately by a write would also work, the correctness of the algorithm does not depend on access to the layer being atomic.

the same time. Wider layers (arrays) provide more parallelism and reduce contention, narrower layers are more likely to be up-to-date. Upon exiting each layer in the funnel (ID in hand), processors first attempt to collide, then advance to the next layer. Implementing layers in message passing is described in [23]. The following presents a high-level step-by-step description of the combining-funnel scheme for a processor p, where initially Location[p] contains the pair $<$ object $= X$, operation $= F >$.

1. **Foreach** funnel layer **do**
   (a) **Swap. Read** q **from** random location in layer, **write** p there.
   (b) **Attempt** to collide with q, **if Succeeded** combine operations.
   (c) **Delay**. Allow some other processor a chance to read p's ID and collide with p. **If** collided behave accordingly.
2. Exit funnel. **Attempt** to perform p's operation on the central object.
3. **Succeeded**? Distribute results. **Failed**? goto 1.

Referring back to our stack example we will show how p and q execute the algorithm. The Location array keeps track of which object a processor is currently operating on, p marks that it is going to apply a PUSH(A) operation on the funnel associated with stack S by setting Location[p] to $<$ object $= S$, operation $=$ PUSH(A) $>$. Let us assume p has read q's ID from a funnel layer at step 1a and now attempts to collide with q. The collision will succeed if both processors are *available* for collision, that is both processors are in funnel S and neither is currently colliding with someone else, this condition can be checked using the Location array. If the collision succeeds, p calculates the combined operation setting the operation field of Location[p] to PUSH({A,B}) and of Location[q] to "WAIT for B to be PUSHed", q is now unavailable for further collisions. In step 1c processors delay to give others an opportunity to collide with them, here q will discover the collision with p and wait for notification that B has been pushed into the stack. Once notified, q will exit the funnel. After passing through all layers, processors access the central object, though they may opt not to wait on a busy lock and instead traverse the funnel again. Once the processor performs its operation on the object, it must deliver results e.g. when p completes its operation on the stack, it informs q that B has been pushed. After passing through all layers they can attempt to acquire the central object. The width of funnel layers decreases with each level since it is assumed that collisions will reduce the number of accesses to subsequent layers. Determining the number of layers to use and the width of each layer is of critical importance and is discussed in Section 3.

2.1 Linearizable Fetch-and-Add and Stack objects

The central object for a combining funnel based fetch-and-add is a location in memory where the current value of the counter is stored. Exclusive access to the counter can be provided using any locking method or through an atomic fetch-and-add primitive in hardware. Fetch-and-add objects support one operation: ADD(x) which atomically adds the value x to the counter and returns its previous value. Combining in fetch-and-add is based on the observation that when two processors want to perform $F\&A(X,a)$ and $F\&A(X,b)$ respectively, if one of them instead performs $F\&A(X,a+b)$ and returns X's current value to itself and $X+a$ to the other – both requests to be satisfied. To facilitate the combining phase we will define another operation, WAIT, which is not a true operation, but rather indicates that a processor is stalled pending the completion of its operation by someone else, namely, its parent. All processors enter the object with an ADD operation. When ADD(x) and ADD(y) collide, step 1b changes the specification of one to ADD($x+y$) and that of the other to WAIT. The processor who is assigned ADD($x+y$) becomes the parent of the stalled processor. To support the distribution phase, parents are responsible for keeping a list of children, holding the identity and request sizes of processors they combined with. Processors in the distribution phase go over the list of children and deliver a result to each of them. Processors who discover they have become children (i.e. the operation field of their element of Location is changed to WAIT) delay in step 1c pending delivery of a result and must then distribute values to their own children.

The most straightforward implementation of a central object for a stack is simply to take a regular serial stack and surround it by a locking mechanism. If two PUSH operations collide we can combine by having one processor, after acquiring the stack's lock, push both values into the stack. By extension, trees of PUSH operations will work the same way, the root performs all the operations once it has the lock on the stack, this will take a time linear in the total number of operations in the tree. We can improve on this naive approach if we make the following observation, if a tree is homogeneous (contains only one kind of operation, either PUSH or POP) then when the the root goes to perform the operations one by one, each operation has a different value of the stack pointer, SP i.e. is done on a different element of the stack. We can therefore view homogeneous trees as a kind of fetch-and-add operation, where the root adds to (or subtracts from, in the case of POP) the current value of SP the size of its tree and delivers to each child an index. Children then continue the process till each node in the tree knows which element of the stack it is supposed to operate on. Any node that receives an index can then immediately perform its stack operation on it. Once the root knows that all operations in its tree are complete, it can release the lock on the stack and allow the next tree to begin operations. This parallel approach reduces the time to complete a tree of operations from linear in the total number of processors in the tree, to linear in the depth of tree, which will usually be much smaller.

What about *combining opposite operations*? Observe that if a PUSH is followed immediately by a POP the stack is returned to exactly the state it had prior to both operations. In a sense adjacent PUSH and POP operations are nothing more than one processor passing a value to another, using the stack as a conduit. Combining can be applied here by having the processor performing PUSH(x) pass x directly to the processor performing POP circumventing the central stack altogether. This *elimination* technique is due to Shavit and Touitou [22]. Here we wish to generalize this approach to handle entire trees of operations, rather than single processors. In doing so we encounter the problem of eliminating trees with different layouts. To avoid the layout problem we will only allow collisions between roots of equal sized trees. Thus, the root of a tree of n PUSH operations may only collide with a similar root, creating a tree of $2n$ PUSH operations, or with the root of a tree of n POP operations leading to

the elimination of both trees. Since all aggregate operations are formed through an identical series of collisions, all are homogeneous and all have the same layout.

2.2 Linearizability

Linearizability is a correctness condition introduced by Herlihy and Wing [13] that allows one to easily reason about and compose concurrent objects. Informally, an implementation of an object is linearizable if we can associate with every operation a moment in time between the start of the operation and its end, and say that the operation appears to have occurred then. We now explain why our fetch-and-add and stack implementations are linearizable. We do so by showing a linearization order consistent with the "real-time" order of operations exists.

Let us imagine a run of the fetch-and-add algorithm in which all combining attempts fail. Processors go through the funnel, swap values on the layer array, but never manage to collide with a partner. In this scenario, each processor carries only its own add request and applies it when it acquires the lock on the central object. This is correct since the linearization order corresponds exactly with the order in which processors acquire the lock. Now let us assume combining does occur, but that when p acquires the lock on the counter it applies the operations in its tree one by one. Notice that since each operation is applied separately and that during this time only p operates on the object, a correct linearization order exists and corresponds to the order in which p applies the operations. In actuality, recall that processors outside p's tree can only examine the object after p releases the lock, so from their point of view it doesn't matter whether p applies all operations at once or one at a time. For processors inside p's tree the distribution phase returns to each of them exactly the same value it would have received had p applied the operations one by one. Thus, for every processor the case in which p applies the operations one at a time is indistinguishable from the case in which all operations are applied at once. The return values which are determined by the distribution phase at each parent implicitly determine the order in which the increment requests are linearized. For example, in our code the linearization order corresponds to pre-order numbering of the nodes of the tree.

The argument regarding linearizability of our stack implementation runs along similar lines, with only the eliminating processors requiring special attention. Consider a pair of processors p and q performing a push and a pop respectively. At some point p learns that it must eliminate with q, at a later point p will write its value for q to read. Since a push/pop pair leaves the stack in exactly the same state we can linearize this pair of operations anywhere between these two points. So long as we do not linearize any operation between them, this will ensure that all processors have a consistent view of the stack. Since there can be only a countable number of operations in the time interval, we can always find a linearization point.

3 Adaption

Combining funnels, like diffracting and elimination trees, are a parameterized data structure: performance of the algorithm is determined by certain parameters which must be chosen and tuned for each application. The number of layers, the width of each layer and the delay at each level can all be optimized based on the expected load on the object and the specifics of the machine being used. As noticed by previous researchers [14, 17, 16] using an adaptive data structure we can provide a solution that dynamically adjusts its parameters based on actual conditions encountered. The number of collisions a processor is involved in at each access to the object can serve as an indicator of the load. Few collisions serve as evidence to low load, and suggest using smaller layers and lower depth. In fact, it may even be possible to avoid the use of a funnel altogether and achieve latency equal to that of a simple locking object. Conversely, many collisions imply wider layers and deeper funnels are needed. Widening layers increases their parallelism as more processors can collide simultaneously. Deepening the funnel increases the number of collisions and reduces the number of accesses to the central object.

We propose the following method for choosing funnel paramaters and applying an adaptive strategy to adapt to encountered load. Using benchmarking techniques determine the set of parameters that gives best performance under maximum possible load. Generally, maximal load occurs when all processors run a program which does nothing but contineously access the object. Note that for our stack object, all processors should apply one type of operation in this test, since otherwise eliminations willl occur, which will reduce "pressure" on the central object. The combining funnel used in the application should be implemented using these "maximal load" parameters which processors can change dynamically in response to load fluctuations. Decisions on parameter changes are made locally by each processor, layers or funnels don't actually grow or shrink, instead processors choose a random layer location from a subset of the full width, or pass through less (or more) layers before trying to gain access to the central object. Figure 3 illustrates two possible adaption strategies. In the one on the left, which better fits our fetch-and-add implementation, processors which believe the load is high enter the funnel at the very top going through all layers, those that believe otherwise can enter the funnel further down and traverse less and narrower layers. On the right hand side is an adaption strategy for the stack. Here processors must always enter on the first level since the level determines the depth of their tree, though if they perceive the load to be low, they can choose to use only part of the layer's width and attempt to access the central object more often.

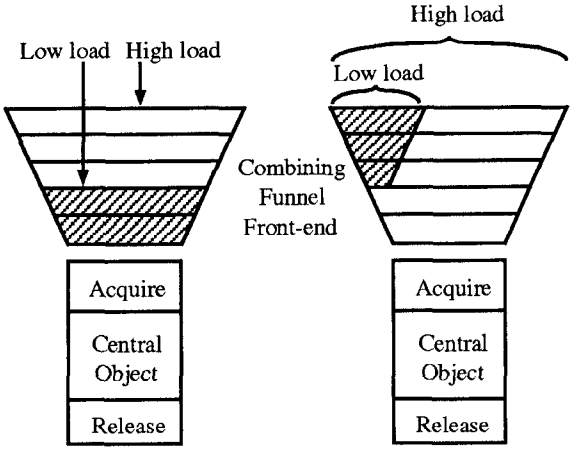

Figure 3: Two methods of adapting layer sizes to different loads. Shaded areas are used when the load is low.

For fetch-and-add the algorithm adapts as follows. Each time a processor p passes through the funnel, it marks l, the number of levels passed through before a collision occurred. Let $\bar{l}$ denote the average of l over N successive operations. Assuming some suitably chosen threshold values T and K, if $\bar{l} < T$, an indication of high contention, p increments a private counter c when c reaches K, p adapts by starting deeper inside the funnel on its next operation. If $\bar{l} \geq T$, an indication of low contention, c is decremented, and when c reaches 0, p adapts by starting at a layer higher up the funnel on its next operation.

For stacks we use a slightly different approach. Each processor keeps a value $0 < f \leq 1$, by which it multiplies the layer width at each level to choose the interval into which it will randomly swap (e.g. if $f = 0.5$ only half the width is used). When a processor p successfully acquires the central object it increments a private counter c, and when c reaches some limit K, f is halved. If p fails to acquire the central object, c is decremented, and when c reaches 0, f is doubled.

A major advantage of combining funnels over tree based methods such as combining trees and elimination trees is that they are substantially easier to adapt to different machine loads. The difference between shrinking a layer array and removing levels from a tree is that processors need not coordinate the move to a different layer size. When using trees, it seems that all processors must agree on the exact size of the tree at all times, otherwise correctness is lost as some processors skip leaves that may be in use by other processors. When using combining funnels, no coordination is necessary, it is possible for processors to have different ideas about layer's width or funnel depth. Each processor makes its decisions locally based on what it perceives the load to be.

3.1 Code for Data Structures

The information kept by each processor performing an operation is logically divided into private and public data. The public data is just the information which processors that collide with this one need inorder to update their own operations. In our implementations only the processor's current operation and status (e.g. is it currently inside funnel x) are made public. Eveything else is private.

Fetch-and-Add Figure 4 lists pseudo-code for our fetch-and-add implementation. We assume per-processor data is accessed through a `my` pointer, and that `obj` and `op` encapsulate data and functions specific to the object and operation (respectively) being performed. Per-processor public data used here is a `Location` word which is used for collisions, and an `Operation` word which points to the public part of a processor's operation. The following is a brief walk-through of the code. Lines 1–5 set up the data structures for the operation. Lines 8–17 contain the collision code, a processor p picks a random layer locations, swaps its `my` pointer for the one written there, q, and attempts to collide by locking itself and q (lines 8–11). If the collision succeeds, q is added to p's list and p continues (lines 12–15). If p discovers that it has been collided with, either immediately or after a short delay, it goes on to wait for a value (lines 17–19 and 21,29). After passing through all layers, p attempts to perform a fetch-and-add operation on the central counter using compare-and-swap (line 23), if this succeeds p moves to the distribution phase. The distribution phase begins with code that waits for a result (lines 32–33), when it arrives p iterates over all processors in its list, handing out a value to each of them by setting their `op->result` field (lines 34-37).

```
Fetch_and_Add( object *obj , operation_type *op )
{
 1    n = 0
 2    subtotal = op->sum
 3    op->result = EMPTY
 4    my->Location = obj
 5    my->Operation = op
 6    while( 1 ) {
 7      for(i=0; i < NumberOflayers; i++) {
 8        r = random() % obj->layerWidth[i]
 9        q = SWAP( obj->layer[i][r] , MYID )
10        if ( CaS(my->Location, obj, NULL ) ) {
11            if ( CaS(q->Location, obj, NULL ) )  {
12               list[n++] = (q , q->Operation->sum )
13               op->sum += q->Operation->sum
14            }
15            my->Location = obj
16         }
17         else goto distribute
18         for(i=0;i<obj->Spin[l]; i++)
19           if ( my->Location != obj) goto distribute
20      }
21      if(CaS(my->Location,obj,NULL)) {
22        val = obj->counter;
23        if(CaS(obj->counter , val, val + op->sum))  {
24          op->result = val
25          goto distribute
26        }
27        my->Location=obj
28      }
29      else goto distribute
30    }
31  distribute:
32    while( op->result == EMPTY) /* spin */;
33    val = op->result
34    for(i=0; i<n; i++) {
35      ( q , qsum ) = list[i]
36      q->Operation->result = val + subtotal
37      subtotal += qsum
38    }
}
```

Figure 4: Code for Fetch and Add implementation

Stack Code for the stack implementation is somewhat more involved though still quite simple. The same public data is used here as before with an additional per-processor public word `Comm` used to relay different types of information between processors. We begin as before with a setup phase (lines 1–4) followed by an attempt to collide (lines 7–10). Two differences bear mentioning: (1) We add the processor's level, `l` to its location marker to make sure collisions only occur between processors on the same level; (2) We allow processors to attempt operation on the central stack not only when they exit the funnel but also once every `obj->Attempts` unsuccessful collision attempts (line 6). If a collision occurs p checks q's operation, if they are the same q is added to p's list and p advances one level (lines 11–14). Otherwise p writes to the `Comm` word of the processor performing the `PUSH` (either p or q) the other processor's `my` pointer (lines 16–20). This way the `PUSH`-ing processor knows the ID of the `POP`-ing processor and can coordinate the rest of the elimination. As before lines (24–29) deal with unsuccessful collisions and delays. If p manages to acquire the central stack (lines 32–47), it first updates the stack pointer and the object's `TICKET` counter, then releases the lock. The routine `update_sp(sp, op, x)` increments `sp` by x if `op` is a `PUSH` and decrements it by x otherwise, in neither case is `sp` allowed to exceed appropriate bounds. The size of the update in line 33 is 2^l since, having been on the l-th layer, p

```
Stack( object *obj , operation_type *op )
{
 1    l = 0
 2    my->Comm = EMPTY
 3    my->Op = op
 4    my->Location = < obj , l >
 5    while(1) {
 6      for(n=0;n < obj->Attempts && l < obj->Levels; n++) {
 7        r = random() % obj->Width[l]
 8        q = SWAP(obj->layer[l][r], my)
 9        if(CaS(my->Location, <id,l>, NULL)) {
10          if(CaS(q->Location, <id,l>, NULL)) {
11            if( q->Op->command == op->command) {
12              my->List[l] = q
13              my->Location = <id , ++l>
14              n = 0
15            }
16            else {
17              if(op->command == PUSH) {
18                 my->Comm = < COLLIDE, q, 0 >
19              else
20                 q->Comm = < COLLIDE , my, 0 >
21              goto collided
22            }
23          }
24          else my->Location = < id , l>
25        }
26        else goto collided
27        for(i=0;i<obj->Spin; i++)
28          if( my->Location != <id, l> ) goto collided
29      }
30      if(CaS(my->Location, <id,l>, NULL)) {
31        if(Acquired(obj->lock)) {
32          sp = obj->SP
33          obj->SP = update_sp(sp, op->command , 1<<l)
34          myticket = obj->TICKET++
35          Release(obj->lock)
36          myplace = update_sp(sp , op->command , 1)
37          for(i=l-1; i>=0; i--) {
38              my->List[i]->Comm = <STACK , my, sp >
39              sp = update_sp(sp, op->command , 1<<i)
40          }
41          while (obj->NOWSERVING != myticket) ;
42          my->Go = 1<<l
43          ret = obj->do_single(myplace,op)
44          Decrement(my->Go)
45          while(my->Go > 0)
46          obj->NOWSERVING ++
47          return ret
48        }
49        else my->Location = < id , l >
50      }
51      else goto collided
52    }
53  collided:
54    while( my->Comm == EMPTY) ;
55    < type , q , val > = my->Comm
56    switch( type ) {
57     case STACK:   /* root acquired SP */
58       sp = update_sp(val, op->command, 1)
59       for(i=l-1; i>=0; i--) {
60         my->List[i]->Comm = < STACK , q, sp >
61         sp = update_sp(sp, op->command, 1<<i)
62       }
63       while(q->Go==0);
64       op->result = obj->do_single(val,op)
65       Decrement(q->Go)
66       break
67     case COLLIDE  /* eliminated and PUSHing */
68       for(i=l-1; i>=0; i--)
69         my->list[i]->Comm = < COLLIDE , q->list[i], 0 >
70       q->Comm = < VALUE , 0, op->data >
71       break
72     case VALUE    /* eliminated and POPing */
73       op->result = val
74    }
}
```

Figure 5: Part 1 of code for stack implementation

knows that it is at the root of a tree of 2^l operations. After releasing the lock, p distributes stack pointer values to all processors in its tree (lines 37–40) much in the same way as is done in the fetch-and-add operation. Now p waits for all pending operations to complete (lines 41) before giving the processors in its tree the "go ahead" to begin by setting its my->Go word (line 42). The my->Go word serves as a double barrier, first barring processors from starting their operations before their tree's turn arrives, then stalling the root until all processors in its tree are done. After performing its own stack operation (line 43) p waits for all operations in its tree to complete (line 45) before incrementing NOWSERVING and allowing the next processor to proceed (line 46). Processors which notice that they have been collided with, jump to the end part of the code. Here they must first wait for updates to their Comm word to reveal the nature of the collision. If the processor which collided with them has captured the central stack (line 57), they must distribute stack pointer values to their children (lines 59–62), and once given the "go ahead" by the root, perform their operation (lines 63–64). In an elimination, first the PUSH-ing processor informs each processor in its list of the identity of a POP-ing processor with which it must eliminate (lines 68–69), then it performs a singleton elimination with its partner (line 70). Eliminated POP-ing processors spin on their Comm word in line 54 till their PUSH-ing partner delivers a value to them.

4 Performance

```
global count, latency

benchmark()
{
   while(count < N) {
     w = random(0, work)
     for(i=0; i<w; i++)
       ;
     start = TIME()
     if ( FETCH_AN_ADD ) {
       a = fetch_and_add(1)
     }
     if ( STACK ) {
       r = random(0,1)
       if(r==0) push(a)
       else a = pop()
     }
     latency += TIME() - start
     count++
   }
}
```

Figure 6: Code for benchmarking fetch-and-add and stack implementations. Global variables are seen by all threads but require no synchronization to access.

Currently, combining trees and elimination trees are the most effective parallel fetch-and-add and stack structures, respectively. We compared combining funnels to these algorithms. We also compared to a simple locking variant of each data structure in order to have a point of reference for performance in low load situations. Our tests were performed on a simulated 256 processors distributed-shared-memory multiprocessor similar to the MIT *Alewife* machine [2] of Agarwal et al. using *Proteus*[2], a multiprocessor simulator

[2] Version 3.00, dated February 18, 1993.

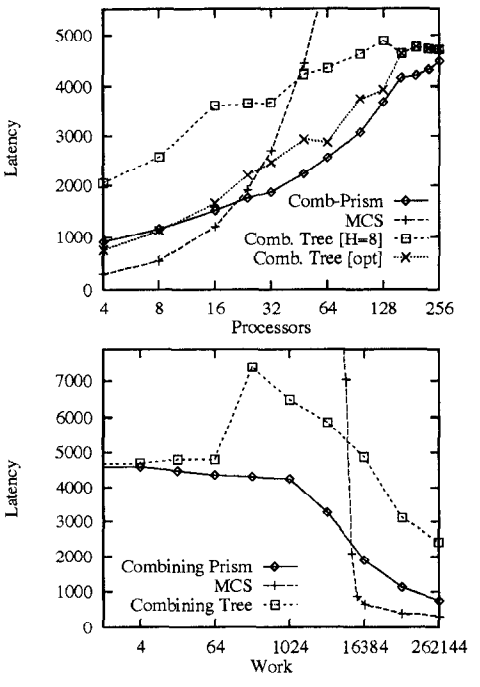

Figure 7: Latency of different fetch-and-add implementations with varying number of processors (*top*) and local work (*bottom*).

developed by Brewer et al. [4, 5]. In our benchmarks, processors alternate between doing local work and accessing the shared object being tested. We ran two sets of benchmarks (see Figure 6), one in which we vary the number of processors and keep local work a small constant and the other in which we vary local work and keep the number of processor at the maximum. In each experiment we measure *latency*, the amount of time (in cycles) it takes for an average access to the object. The span of the work parameter tested represents the full range of access patterns, from maximal load were processors spend all their time in the data structure (`work=0`) to minimal load were the time spent inside the data structure is less than 1% of the time spent spinning in work loop. Funnel parameters were chosen as described in Section 3 (that is, 256 processors and work was set to 0) and adaption threshold values T and K were chosen as those which gave the best performance (from a list of values which seemed logical) for 64 and 16 processors. For funnels, adaption is incorporated into the implementation of the object and the overhead it entails is counted in the latency of the operations.

Fetch-and-Add performance. The graph in the left part of Figure 7 shows the performance of fetch-and-add implementations as the number of processors changes and local work is a small constant. We plotted two curves for combining trees, one in which the height of the tree is optimal (marked `H=opt`) i.e. for p processors a tree of height $\lceil \log p/2 \rceil$ is used. The other curve is of a tree of constant height eight (marked `H=8`), needed to support 256 processors – the maximum number of processors in our simulations. The curve marked `MCS` represents performance of a single counter protected by an MCS-lock [19]. The graph shows that combining funnels are only substantially more expensive than the MCS-lock for eight or fewer processors, at sixteen processors both methods perform about equally, beyond this level of concurrency latency of the MCS-lock increases rapidly and it becomes totally unusable beyond 48 processors. Combining funnels outperform optimal height combining trees by a small amount for all levels of concurrency. Notice, that this result indicates that even if one could dynamically adapt between different tree sizes [16], perfectly choosing the correct size tree every time and with no overhead (such as measuring load or synchronizing the move to a new sized tree), performance would still not be as good as when employing adaptive funnels. The power of adaption is evidenced by examining the curve for constant height combining tree. When using 64 processors, the combining tree is two levels too deep and has 70% higher latency than the funnel, halve the number of processors and the tree becomes three levels too deep and twice as slow as our method. In the right-hand graph of Figure 7 256 processors are used with varying local work. The combining tree had height eight even though the tree is unlikely to reach that level of concurrency. This has a very adverse effect on the amount of combining in the tree and results in a "spike" in the latency curve. This is the worst possible scenario for combining trees since processors that do not combine are essentially locked out of the path to the root. No such sudden increase appears in the latency curve of combining funnels, since no predefined "tree" structure exists and paths cannot be locked. The situation where processors are constantly arriving "too late" to combine and must wait for their would-be partners to ascend and then descend the entire tree does not arise. If there are many processors in the data structure chances of colliding are good since combining can occur between any pair of processors. As concurrency drops the shrinking layer width helps keep chances of colliding high while the shrinking depth lowers latency. Comparison with the MCS lock shows that that method is only applicable for very sparse access patterns i.e. for rarely used objects, however, under such circumstances it can be up to three times as fast as combining funnels.

Stack performance. In [22], Shavit and Touitou compare different stack implementations and the one based on elimination trees is shown to outperform the rest. We compared our stack implementation to the non-linearizable elimination tree and to two linearizable methods: a serial stack protected by an MCS lock and a combining tree based stack. We found that combining trees are always substantially slower than combining funnels, 10 times slower at maximum load. This is mostly due to elimination though we found that even if elimination is not used e.g. all operation are `PUSH`, trees were three times slower than funnels. For this reason we do not display these results in our graphs, concentrating on other methods instead.

In Figure 8 we again see that the latency of simple MCS based locking is unsurpassed at low concurrency levels, but adding more processors or reducing local work render the method impractical. At the opposite end of the spectrum, at 256 processors elimination trees outperform combining funnels by about 10% (the parameters used by both methods have been optimized for this case). However, the latency curve for elimination trees has a downward slope, indicating an *increase* in latency as processors are removed from the simulation (this is consistent with the results of [22]), while the curve for combining funnels slopes up – fewer processors mean lower latency. Thus at 64 processors the difference

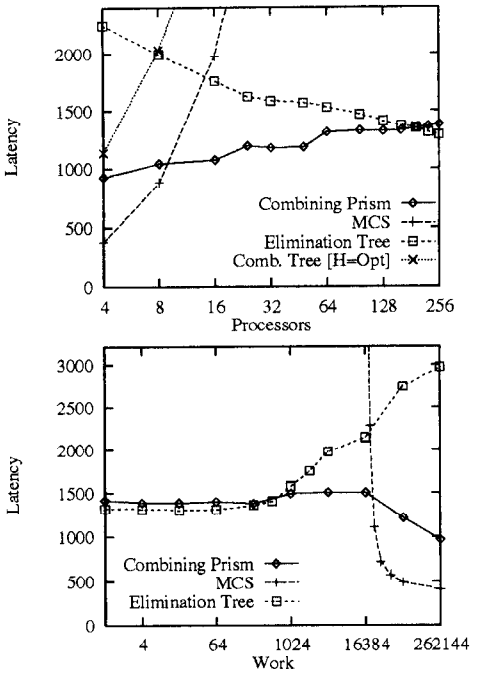

Figure 8: Latency of different stack implementations with varying number of processors (*top*) and local work (*bottom*).

is 16% and at 32, 34% in favor of combining funnels. The graph for varying local work tells a similar story, initially elimination trees have a slight edge in performance, but at around the middle of the graph[3] the curve begins to slope upwards. The explanation lies in the inability of elimination trees to adapt their height. As concurrency drops so do chances of diffraction, thus processors are forced to descend further down the tree before either eliminating or storing their element at the leaves.

5 Discussion

Combining funnels are a generalized framework for developing highly concurrent data objects. We have shown how they enable taking simple fetch-and-add and stack objects, and by following a structured step-by-step approach, creating effective parallel data structures.

Currently all our experiments were done by simulation. However, machines large enough to benefit from these methods are slowly becoming more common. We hope to be able to try these methods out in a real world setting in the near future. We are currently looking for a large scale application into which we might "plug-in" our methods and see if performance has really improved. Also of interest are composite data structures made up of several smaller objects, some implemented using combining funnels.

6 Acknowledgments

We would like to thank Dan Touitou for his many helpful comments.

[3] At this point the processors are spending about half their time doing local work and the other half updating the stack.

References

[1] A. Agarwal and M. Cherian. Adaptive Backoff Synchronization Techniques. In *Proceedings of the 16th International Symposium on Computer Architecture*, pp. 396–406, May 1989.

[2] A. Agarwal, D. Chaiken, K. Johnson, D. Krantz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Maa, and D. Nussbaum. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Scalable Shared Memory Multiprocessors*, Kluwer Academic Publishers, 1991. Also as MIT Technical Report MIT/LCS/TM-454, June 1991.

[3] 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.

[4] E.A. Brewer, C.N. Dellarocas. PROTEUS *User Documentation*. MIT, 545 Technology Square, Cambridge, MA 02139, 0.5 edition, December 1992.

[5] 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.

[6] G. Della-Libera and N. Shavit. Reactive Diffracting Trees In *Proceedings of the 9th Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1997.

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

[8] J.R. Goodman, M.K. Vernon, and P.J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 64–75, April 1989.

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

[10] A. Gottlieb, B.D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[11] G. Graunke and S. Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *IEEE Computer*, 23(6):60–70, June 1990.

[12] M.P. Herlihy, B.H. Lim and N. Shavit. Scalable Concurrent Counting. *ACM Transactions on Computer Systems*, 13:4, (1995) 343-364.

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

[14] A. Karlin, K. Li, M. Manasse and S. Owicki. Empirical Studies of Competitive Spinning for A Shared Memory Multiprocessor. In *13th ACM Symposium on Operating System Principles (SOSP)*, pp. 41–55, October 1991.

[15] C.P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization on multiprocessors with shared memory. In *Fifth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1986.

[16] B.H. Lim and A. Agarwal. Reactive Synchronization Algorithms for Multiprocessors. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pp. 25–35, 1994.

[17] B.H. Lim and A. Agarwal. Waiting Algorithms for Synchronization in Large-Scale Multiprocessors. In *ACM Transactions on Computer Systems*, 11(3):253–294, August 1993.

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

[19] 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.

[20] G.H. Pfister et al. The IBM research parallel processor prototype (RP3): introduction and architecture. In *International Conference on Parallel Processing*, 1985.

[21] 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.

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

[23] N. Shavit and A. Zemach. Diffracting Trees. *ACM Transactions on Computer Systems*, 14(4), pp. 385-428,Nov 1996.

[24] P.C Yew, N.F. Tzeng, and D.H. Lawrie. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.