Combining Funnels: A Dynamic Approach To Software Combining *

Nir Shavit†
Department of Computer Science,
Tel-Aviv University,
Tel-Aviv 69978, Israel.

Asaph Zemach‡
Department of Computer Science,
Tel-Aviv University,
Tel-Aviv 69978, Israel
and
Tera Computer Company,
411 First Avenue South,
Seattle, WA, 98104.

July 6, 1999

Abstract

We enhance the well established software combining synchronization technique 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. On shared memory multiprocessors the new method allows to dynamically build combining trees with depth logarithmic in the actual number of processors concurrently accessing the data structure. 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 Touitou) to the funnel structure, the randomly constructed "tree" is transformed into a "forest" of disjoint (and on average shallower) trees of requests, 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 outperforms combining trees of fixed height by as much as 70%. In fact, even compared to combining trees optimized for a given load, funnel performance is the same or better. Elimination trees, which are not linearizable, are 10% faster than funnels under highest load, but as load drops combining funnels adapt their size, giving them a 34% lead in latency.

*A preliminary version of this work appeared in Principals of Distributed Computing (PODC98).

†Supported by NSF grant CCR-9520298, A grant from the Israel Ministry of Science, and a grant from the Israel National Academy of Sciences. Contact Author: E-mail: shanir@math.tau.ac.il

‡Supported by an Israeli Ministry of Science Eshkol Scholarship.

1 Introduction

When different threads running a parallel application on a shared memory machine access the same object simultaneously, a synchronization protocol must be used to avoid interference. Since modern shared memory 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.

1.1 Software Combining

It is well documented [1, 8, 14, 26] 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" [8, 26] contention is *combining*. Combining was invented by Gottlieb et al. to be used in switches of a network which connects processors to memory [10, 25]. It allows one 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 (Gottlieb et al. [11]), hardware switches can perform combining on several different kinds of messages, including reads, writes and fetch-and-add operations [18]. The most notable example of combining in software are the combining trees of Goodman et al. [9] and Yew et al. [33] for performing fetch-and-add. 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, which in turn propagate it down the tree.

Unlike faster lock-free structures such as the Diffracting trees of Shavit and Zemach [29], combining trees are linearizable. Linearizability is a consistency condition defined by Herlihy and Wing [15], which allows the programmer to treat complex operations as if they happened atomically. Dealing with linearizable data structures makes programming easier since one can assume, for example, that once an update is complete (e.g. the call to the update procedure returns) all later operations consistently see its effects.

Combining trees are widely applicable and can be used to enhance the implementation of any fetch-and-Φ [11, 18] synchronization primitive, as well as some simple data structures such as sets. In the classic combining tree scheme, 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. In the classical combining scheme, processors must acquire this lock in order to ascend from a node to its par-

ent. Thus, the number of processors that may concurrently enter a node is limited to two regardless of the load, effectively eliminating contention. (Though it is possible to construct trees with fan-out greater than two in order to reduce tree depth, that would sacrifice the simplicity of the nodes and increase the number of instructions required to traverse them. Indeed, preliminary experiments which we conducted on such trees show that the "higher fan-out for lower depth" trade-off is not worth it: the overhead of more complex inner nodes overshadows the benefits of decreased depth).

1.2 Combining Under Varying Loads

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

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. [14], 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 and in [29]).

The original software combining tree algorithm of Goodman et. al. does not offer easy ways to work out these difficulties: they all introduce performance tradeoffs. For example, pipelining requests up the tree (by removing locks or allowing processors to overtake locked nodes) introduces a tradeoff between node latency and the level of overtaking. Naive attempts to allow overtaking at locked nodes would mean that other nodes (especially the root) could be reached by many processors at a time. The need to have each node handle this increased parallelism and contention effectively complicates the protocol used in the tree nodes and increases latency.

Combining trees are also not easily amenable to an adaptive strategy which shrinks the tree when average load is low (e.g. the reactive locks of Lim and Agarwal [19]), since there is no clear way to lower the number of nodes and at the same time 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 [7]) tend to be complex and require significant tuning efforts.

This is not to say that there are no known adaptive combining schemes. Gupta and Hill [13] and Mellor-Crummey and Scott [24] have devised adaptive combining tree barriers that support changing the layout of a combining tree on the fly. However, their constructions cannot be readily applied to implement general data structures. The reason is that in order to work they require two key assumptions that cannot be met by general data-structures. First, they require that all processors arrive at the data structure and traverse it exactly once before it is reused, and second, that the information passed between processors be exactly the same ("barrier complete") and not unique ("your result is x"). Both conditions can be met only because of the very nature of a barrier – everyone is delayed until the current instance of the barrier is complete.

In summary, it seems that allowing pipelining of requests, eliminating unnecessary waiting, and modifying the tree algorithm to be adaptive, would benefit performance. However, the associated tradeoffs suggest that there is no easy way to fit all these properties into the classic combining tree framework.

1.3 Beyond trees: the new approach

In this paper we present a new general method for implementing the "combining" paradigm on large-scale shared memory multiprocessors. Our method, *combining funnels*, replaces the "static" tree with a collection of randomly created dynamic trees. This is done via a cascade of *combining layers* through which requests are funneled and combined to form the dynamic trees, hence the name combining funnels. It allows us, especially under high loads, to reap many of the benefits of combining without the drawbacks of using a global tree structure.

In broad terms, the combining funnel approach can be stated as follows. Assume you are given a simple data object with *combinable* [10, 18] operations, and that this object 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 (see Figure 1). All processors attempting to access the object pass through the layers of the funnel and can at each layer 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 "central" object.

The combining layer structure provides the basis for an adaptive combining structure. Adaptive algorithms, allowing the data structure to change behavior to accommodate different access frequencies, have been used both in locking (see Karlin et al. [17] and Lim and Agarwal [20]) and for more general fetch-and-Φ operations [19]. The work of Lim and Agarwal [19] 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 single lock to a full funnel. Unlike the approach of Lim and Agarwal, adaption does not require global agreement as to the size of the data structure. Rather, each processor dynamically and independently chooses the part of the funnel it will try and traverse. This lack of coordination among processors lowers overhead and simplifies the protocol. It means however that at various points in the execution, different processors might end up with different size decisions: some using large funnels, some small. Nevertheless, this does not affect the algorithm's correctness, and as we will show, produces a significant performance advantage.

Apart from being adaptive, combining funnels also support a mechanism for *elimination*, as defined in the work of Shavit and Touitou [27]. Elimination is used in shared objects that support operations with *reversing semantics*: applying the operations to the object in a certain order leaves the object unchanged. A stack is good example: applying a push operation, immediately followed by a pop operation, returns the stack to its original state. A pair of operations with reversing semantics can be eliminated if both requests can be satisfied, that is, correct values are returned without any update to the central object. The combining funnel allows us to extend elimination from pairs of operations to dynamically constructed "trees of operations" – satisfying complete trees of operations in parallel without accessing the central object.

In this paper we present 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 [15], that is, operations on them appear to be atomic. This is a property found in combining trees but not in previous prism based methods such as diffracting trees and elimination trees.

We evaluate the performance of the new 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 256 processor shared memory multiprocessor similar to the MIT Alewife machine [2]. Our simulation uses the well

established Proteus simulator [4, 5]. Since the type of futuristic applications that will benefit from such high levels of concurrency are currently not available, we use a standard collection of synthetic benchmarks that mimic their possible access patterns. Based on our empirical results, we believe our linearizable fetch-and-add and linearizable stack objects display the kind of performance, robustness and simplicity which would make them useful additions to the parallel computing tool-boxes of the near future.

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 of the data structures implemented.

2 Combining Funnels

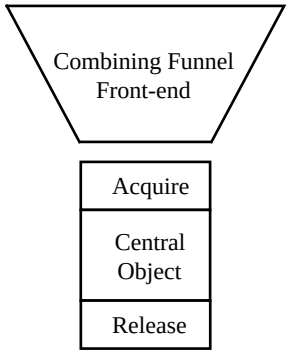


Figure 1: Schematic depiction of combining funnel mechanism

We first present our combining-funnel scheme in a generalized form and then show how both a fetch-and-add and a stack object fit into the framework. The idea, illustrated in Figure 1, 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. A more elaborate example appears in Figure 2.

In a shared memory environment, a funnel layer can be implemented using an array.[1] A processor arriving at the array picks a location at random and applies a register-to-

[1] An algorithm that can be modified to implement funnel layers using message passing is described

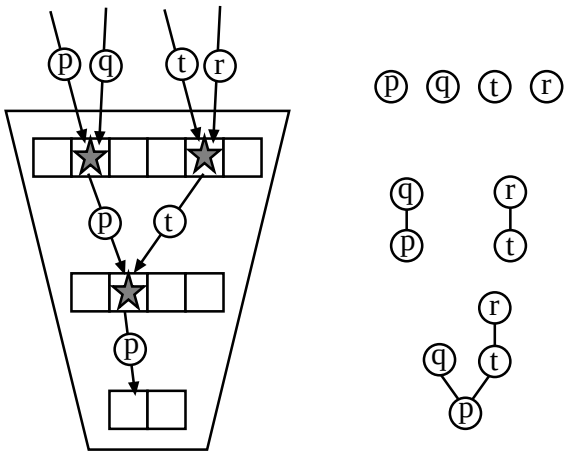


Figure 2: Example of processors going through a funnel. On the left we see p, q, r and t as they go through the first layer where p collides with q and t with r, then q and r wait while p and t advance to the second layer where they collide. On the right side we see how combining trees are dynamically created by collisions, the waiting processor becomes the child of the advancing processor.

memory-swap operation on it,[2] 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 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. Notice that a processor only ever waits if it has successfully combined with another processor that is going to perform its operation for it. A processor is never delayed simply because an array location is "in use." This is not the case with combining trees where tree nodes can be locked, thus preventing the advance of unrelated operations.

2.1 An overview of the algorithm

The following presents a high-level step-by-step description of the combining-funnel scheme for a processor p. The algorithm makes use of a `Location` array, with one element per processor. Each element has two fields: `object` which is the name (or address) of the object p is trying to operate on; and `operation` which is the operation p is trying to apply to the object. For an object X and an operation F, 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 collide**(p,q). **If Succeeded combine**(p,q). Possible combining results include: p performing both operations while q waits for notification; and elimination of both operations.

in [29].

[2] 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.

(c) **Delay**. Allow some other processor a chance to read p's ID and collide with p. **If** `Location[`p`].operation` has changed, perform the new operation. Usually this is either a wait for notification, or an exit of the funnel (if eliminated).

2. Exit funnel. **Attempt** to perform `Location[`p`].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. Processor 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 routine **collide**(p,q) actually tries to perform the collision. The collision will succeed if both processors are operating on the same object: `Location[`p`].object = Location[`q`].object`; and `Location[`p`].operation` is combinable with the operation `Location[`q`].operation`. The decision regarding which operations are combinable is object dependent. In a stack, any pair of colliding operations can be combined. A processor that has already collided and is now waiting for its operation to be performed by some other processor, cannot be further combined.

If the collision succeeds, p calculates the combined operation using **combine**(p,q). **Combine** updates the operation field of both `Location` array elements to reflect the results of the collision.[3] In our stack example, when two `PUSH` operations collide, `Location[`p`].operation` is set to `PUSH({A,B})` and `Location[`q`].operation` is set to "`WAIT for B to be PUSHed`." Processor 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 both the funnel and the stack object.

After passing through all layers, processors can 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. 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. We now present our two combining funnel based data structures: fetch-and-add and stack.

2.2 Fetch-and-Add

To assist in presenting the fetch-and-add structure, we introduce the following high-level description of the stages that one must go through to add a combining-funnel front-end to an object. The step numbering refers to the algorithm in Section 2.1.

1. Decide what the object is and what operations it supports.
2. Determine when two operations can be combined and what the result is (step 1b). The combined operation may sometimes imply creation of new operations not defined in the original object specification. In this case, the semantics of these operations (for step 2) and their interaction with the original ones must be defined.
3. Determine what actions processors must take when they discover they have been combined with (step 1c).

[3] In this informal description there appears to be a race condition between **collide** and **combine**. The pseudo-code given in the appendix solves this using compare-and-swap. Locking of the elements of the `Location` array can also be used to achieve this effect.

4. Determine if a distribution of results is necessary and how it should be done (step 3).

Using this methodology we now construct a combining funnel for a fetch-and-add object. The central object for a combining funnel fetch-and-add is a location in memory where the current value of the counter is stored. Exclusive access to the counter can be supplied 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 following observation. 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. Detailed pseudo-code for the fetch-and-add object can be found in Appendix A.1.

We can distinguish between two types of information associated with each fetch-and-add operation: public and private. The public part is held in the `Location` array and includes the address of the fetch-and-add object and the operation `ADD(`x`)`. The private part is each processor's list of children. To see why this list can be kept private consider a collision between a processor p performing an `ADD(`x`)` operation and a processor q whose operation is `ADD(`y`)`. If p is to continue with an `ADD(`$x + y$`)` operation, it must know that part of the result of this operation must be distributed to q. However, if r now collides with p, it is enough that r knows that p's operation is `ADD(`$x + y$`)`; it does not need to know that p intends to distribute part of the result to q. This is why we can say that p's operation becomes `ADD(`$x + y$`)` and not e.g. "`ADD(`x`) for` p `and ADD(`y`) for` q." Figure 3 illustrates the data associated with each of the fetch-and-add operations and shows how it is updated during a collision. We see that P_4's `ADD(4)` operation is actually an aggregate of its original operation and the `ADD(1)` of P_5: we can deduce that P_4's original operation was `ADD(3)`. When P_1 collides with P_4 it updates its own data, both private and public. It also updates the public part of P_4's data. Only the information available in P_4's public data is used by P_1 in the update.

Collisions between processors dynamically create combining trees as the processor who initiated the collision becomes the parent of the processor with which it collided. The trees created are not necessarily binary since a single processor may collide with many processors, each of which might have collided with any number of other processors. Figure 3 shows P_4 as the parent of P_5, and P_1 as the parent of P_2 and P_3. Figure 4 illustrates the tree created when P_1 collides with P_4 and becomes its parent. We also see other parent-child relationships (e.g. P_3 is the parent of P_6) which can not be deduced from the lists held by P_1 or P_4. If P_1 now manages to acquire the central counter, it will increment it by 16, the sum of all increment requests in the tree. Assuming the central counter held the value 0 when it was acquired by P_1, the figure shows what value will be returned by each processor.

The random nature of the collisions in the fetch-and-add implementation means the trees have no predefined shape. Consider a tree t with depth d containing n operations.

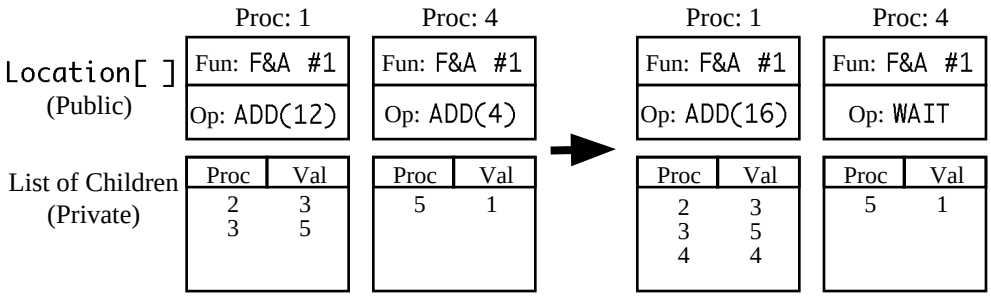
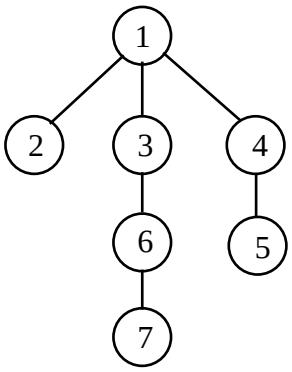
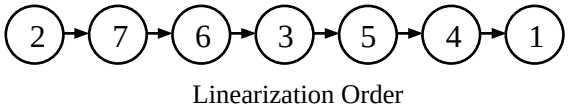


Figure 3: Example of data structures updated during a collision between two fetch-and-add operations. Visible are the two fields of the `location[]` array: `object` in this case "fetch-and-add object #1"; and `operation` in this example either `ADD()` or `WAIT`.



Processor	Original Request	Final Request	Return Value
1	4	16	12
2	3	3	0
3	2	5	6
4	3	4	9
5	1	1	8
6	1	3	5
7	2	2	3



Linearization Order

Figure 4: *left:* Dynamic combining tree created by collisions; and *right:* A table showing how values are propagated up and down the tree and the linearization order determined by the return values.

If $n = d$ the tree can actually be a linear list of operations. This is the worst case in terms of the time it takes to distribute results to children. However, notice that the chances of this happening are exponentially small. If a list shape tree t collides with another tree t', the only way for the new tree to still be a linear list is if t' is a single operation and t' becomes the root. The probability of t always becoming the child in a collision is exponentially small in the number of collisions.

2.3 Stacks

This section describes the implementation of a stack object. 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, as we have done in Figure 5. Standard stack operations are `PUSH`(x) which pushes x unto the top of the stack and `POP` which returns the topmost value in the stack. How can we apply a combining funnel here? 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

```
counter SP
data_type Stack[STACKSIZE]
lock StackLock

int push(datatype d)
{
  Acquire(StackLock)
  if( SP == MAXSIZE )
      result =  STACK_FULL
  else {
      Stack[SP] = d
      SP++
      result = PUSHED
  }
  Release(StackLock)
  return result
}
```

```
datatype pop()
{
  Acquire(StackLock)
  if( SP == 0 )
      result =  STACK_EMPTY
  else {
      SP--
      result = Stack[SP]
  }
  Release(StackLock)
  return result
}
```

Figure 5: Simple stack implementation

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. Colliding `POP`s are analogous. There is some gain here since we amortize the time it takes to do lock operations over many stack operations, though it is not as significant as in the fetch-and-add case.

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 performs the operations one by one, each operation has a different value of the stack pointer (`SP`). In other words, each operation is performed 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 the tree, which will usually be much smaller.

2.3.1 Elimination and the layout problem

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, `PUSH` and `POP` operations that immediately follow one another are nothing more than one processor passing a value to another, using the stack as a conduit. Such operations are said to have *reversing semantics* since one reverses the affects of the other – when applied in the right order. *Elimination* [27] is a technique which allows the pairing off of operations with reversing semantics, such that both processors exit the data structure with correct results, but the object itself is not updated. For example, p performs `PUSH`(x) while q performs `POP`. The operations collide and are eliminated: p passes the value x directly to q. Processors p returns with an indication of success, q returns with x as its popped value. Though the stack was never touched, the result is indistinguishable from the case in which p actually pushed x unto the central stack, and q retrieved it from the stack.

We wish to generalize elimination to handle entire trees of operations rather than single operations. Thus, when a tree of push operations collides with a tree of pop operations, values are passed from one tree to the other, and the central stack is avoided. However, if colliding trees have a different layout, this can be a problem. Consider the situation illustrated in Figure 6, where a tree of three `POP` operations collides with a tree of five `PUSH` operations. The most natural thing to do is to transfer three elements from the "push-ers" to the "pop-ers" thereby eliminating the left tree altogether. However, it is not clear how to decide which "push-er" should be paired with which "pop-er." It would seem that we have to engage in some "layout-matching-protocol," which would most likely increase latency.

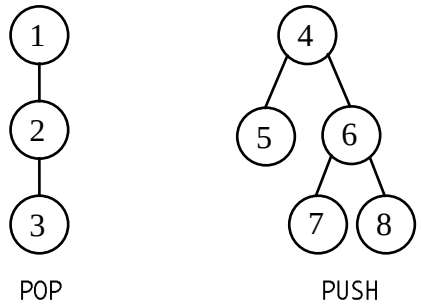


Figure 6: Collisions between trees of different layouts can make elimination difficult.

2.3.2 Solving the layout problem

To avoid the layout problem we choose to only allow collisions between roots of trees that have the same number of operations. Thus, the root of a tree of n `PUSH` operations may only collide with a root of a similar tree, creating a tree of $2n$ `PUSH` operations. It can also collide 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. Limiting collisions to trees of equal size forces all trees to be isomorphic. For example, a tree of eight operations is always formed by the collision of two singleton operations, followed by a collision with a double operation and ending in a collision with a quadruple operation as in Figure 7. To increase the chance that collisions occur between trees of the same size we alter the method by which processors move from one layer to the next. All processors enter the combining-funnel at the first layer (as usual) but advance to the next layer only after a successful collision, if passing through a layer does not yield a collision, processors remain at the same layer. This way all processors spinning at the i-th layer are roots of trees with 2^i operations.

However, this introduces the possibility of starvation. To avoid starvation, processors periodically attempt to perform their operations on the central object regardless of which layer they are on. Eliminating trees of the same size and layout is substantially easier than doing so for general trees as illustrated in Figure 8. If the trees are singleton operations elimination is performed by having the "push-ing" processor hand its value directly to the "pop-ing" processor. Otherwise, let p and q be colliding roots of trees of size 2^l. Note that both p and q each have l children and that the i-th child of each is the root of a tree of size 2^i. First, p sends q (a pointer to) a list of its l children, then q sends its i-th child the name of p's i-th child, now for all $i \in \{1 \dots l\}$ the i-th child of q eliminates with p's i-th child, finally p and q perform a singleton elimination. Note that if the children that are roots of larger trees are informed of

their partners first, we can expect all collisions to occur within time that is logarithmic in the size of the tree. Figure 8 shows that the time to eliminate a tree of l levels is $T(l) = \max\{2 + T(l-1), 3 + T(l-2), \ldots, 1 + l + T(0)\} = 2l + 1$.

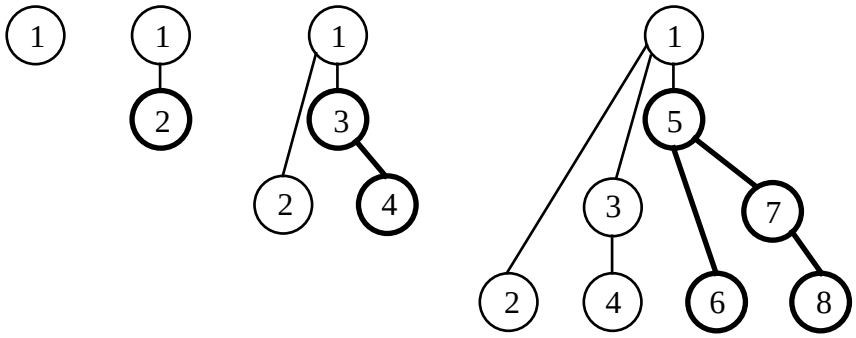


Figure 7: Forming aggregate operations by collisions of equal size trees.

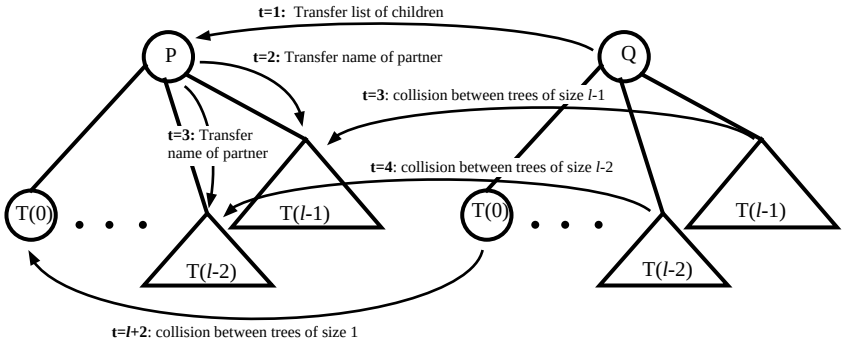


Figure 8: Elimination of trees with equal size and layout is simple.

Earlier we said that a root acquiring the lock on the central stack releases it only after all operations in its tree are done. However, we found that better performance can be achieved by pipelining, the first stage of the pipe is the update to `SP` and the second is the update of the `Stack` array. The pipeline is implemented by using two locks, the first on `SP` and the second, a ticket lock [22], on `Stack`. When a root acquires the first lock it updates `SP` and receives a ticket for the second lock, then it releases the first. This assures that operations on `Stack` occur in exactly the same order as operations on `SP` which maintains the pipeline. The reason we require such a strict order is that if we allow operations to overtake one another in the pipeline the resulting stack would not be linearizable. A detailed discussion of why both fetch-and-add and stack implementations are linearizable appears in Section 2.4.

2.3.3 The Push and Pop operations

We present a high level description of a `PUSH` operation for processor p going through funnel s, the `POP` operation is analogous. The implementation uses four operations: `PUSH` and `POP` with the obvious semantics, `WAIT` which is returned to one of the processors when two of the same type of operation collide, and `ELIMINATE` which is returned to both processors at a collision between opposite operations. As usual when two `PUSHes`

or POPs collide, one becomes the parent (the root, in fact) and retains its operation, and the other, the child, has its operation changed to WAIT. Pseudo-code for a push is:

1. Location[p] $\leftarrow <$ funnel $= S$, operation $=$ PUSH(A) $>$. $l \leftarrow 0$.
2. Swap. Read q from random location in layer l, write p there.
3. If q's size = my size, attempt to collide with q.
4. Collided? If q's operation is PUSH, change q to WAIT, add q to list of children, $l \leftarrow l + 1$. Otherwise change q to ELIMINATE and eliminate both trees.
5. Delay. Give some other processor a chance to collide with me.
6. Collided with? If operation is changed to ELIMINATE, eliminate. Otherwise, WAIT for root to either eliminate or acquire the central stack.
7. $i \leftarrow i + 1$. If i MOD CONST $= 0$ attempt to acquire lock.
8. Acquired?
   (a) Copy MYSP$\leftarrow$SP, MYTICKET$\leftarrow$TICKET, increment SP by 2^l, increment TICKET by 1, release lock.
   (b) Distribute stack locations to children.
   (c) When NOWSERVING reaches MYTICKET give "go ahead" for all processors in the tree to push data into the stack.
   (d) When all processors in my tree are done, increment NOWSERVING. Exit.
9. Didn't acquire. goto 2.

A processor that collides with a like operation and becomes the child must WAIT till it learns the fate of its tree (step 6). If the tree's root enters the central stack update pipeline it will be given a location in the stack on which to operate and must then wait for the tree to reach the second pipeline stage (update Stack) before acting. Once the root enters the second stage it informs the children, and they all push their values into the stack, each child that completes its stack operation decrements a counter stored at the root, when this counter reaches zero the root knows that all operations in its tree are done and it can safely advance the ticket counter. If the root collides with an opposite operation both trees are eliminated.

2.4 Linearizability

This section introduces Linearizability and then explains why our constructed funnel based data structures are linearizable.

Linearizability is a consistency condition introduced by Herlihy and Wing [15], that allows one to easily reason about and compose concurrent objects. Informally, an implementation of an object is linearizable if operations on it appear to be atomic to all processors. More formally, assume that a data structure like a counter or a stack is given a sequential specification. By this we mean that it is described as an abstract data type whose behavior is determined by a collection of allowable sequential executions. For a stack each such execution would be a sequence of push and pop operations and their returned values. An implementation of a data structure is said to be *linearizable* if for every concurrent execution, that is, one in which operations may overlap in time, one can:

- associate with every implemented operation a single time point within its actual execution interval at which the operation is said to take place,
- such that the associated execution is a valid sequential execution in the data structure's sequential specification.

Put differently, Linearizability means that by compressing an operation's actual execution interval in time to a single point (thus preserving the real-time order), we can place all operations on a single time-line, and the resulting totally ordered execution then conforms to the object's sequential specification.

We now explain why our fetch-and-add and stack implementations are linearizable, which amounts to showing a linearization order. The interested reader can formalize these proof outlines in the I/O automata model of Lynch and Tuttle [21].

2.4.1 Linearizability of Fetch-and-Add

To understand why our fetch-and-add construction is linearizable, let us first imagine a run of the 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. Let S be the set of processors whose operations are in p's tree. Let t_i^0 for $i \in S$ denote the starting time of i's operation. Similarly, t_i^1 denotes the end time. Also, let a denote the time at which p acquires the object. Clearly $t_i^0 \leq a \leq t_i^1$ for all i, since no operation in S completes until p acquires the lock. Let p_i denote the time at which p performs the operation of processor i. We get $t_i^0 \leq a \leq p_i \leq t_i^1$. Thus each operation is linearized correctly.

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. Using the notation of the previous paragraph, let a be the time at which p applies the operation to the object, and b be the time the next operation is applied to the object. We number the operations in p's tree in pre-order, such the operation i is numbered $n(i)$. The linearization order of the operations in S is therefore: $a \leq t_0 < t_1 < \ldots < t_n \leq b$, where operation i is considered performed at time $t_{n(i)}$.

2.4.2 Linearizability of the Stack

The argument regarding Linearizability of our stack implementation runs along similar lines. First we look only at operations which don't combine or eliminate. These all end when the processors performing them acquire the main lock on the stack, so they have a natural linear order. Now we allow processors to combine operations, but ignore the fact that the implementation splits the stack pointer increment phase from the actual stack insertion. This gives us a natural linear order among trees, and since no other processor may probe the stack until all operations in a tree are complete, we only need to concern ourselves with tree members when assigning a linearization order. As noted earlier a property of homogeneous trees is that stack locations distributed to the processors are monotone and each processor is assigned a different element. This yields a linearization order based on corresponding stack locations, e.g. for two pushing processors the one with the lower location is linearized first. The fact that we split the update of the stack

pointer from the update of the stack itself is of no consequence since we use a ticket lock to make sure that access to the stack is in exactly the same order as access to the stack pointer.

Having established a linearization order among operations which end up being applied to the stack itself, we need to account for eliminating processors. The recursive nature of the elimination process allows us to ignore the fact that we eliminate whole trees and simply look at pairs of eliminating processors. Consider a pair of processors p and q performing a push and a pop respectively. At some point p knows that it must eliminate with q (either p performed the elimination, or its parent told it), later 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 anywhere in the time interval in which their operations overlap. As 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

We begin with a general discussion of parameterized data structures and adaption, and later outline the specific strategies we use in our implementation.

3.1 Parameterized Data Structures

Combining funnels, like diffracting and elimination trees, are a parameterized data structure: performance of the algorithm is determined by certain parameters which differ from application to application. For funnels, these are the number of layers traversed, the width of each layer and the delay at each level. Each of these can be optimized based on the expected load on the object and the specifics of the machine being used. Contention at the central object can occur if there are too few collisions. This might be a result of layers that are too wide, too few layers, or delay times that are too short. Funnels that are too deep and overly long delay times increase the latency of each operation, whereas narrow layers cause contention on layer locations. Clearly the right choice of parameters is of paramount importance to achieving best possible performance.

The parameters however, differ from one application to the next, and for the same application as the load on it changes. Consider for example a fetch-and-add object. If this object is accessed only rarely, the best performance would be achieved by implementing it using a single location in memory protected by simple, low overhead locking method. However, this implementation works extremely poorly for frequently accessed counters [3, 12, 22]. Other implementations have the opposite behavior, they perform well when accessed frequently but their overhead is prohibitively high for rarely accessed objects [3, 22].

3.2 Adaptive Structure of Funnels

From the above examples it becomes clear that tuning the structure, that is, optimizing its parameters for each application and load, is not a feasible solution. The solution is to use an *adaptive* strategy for automatically tuning the parameters to the current load on the data structure. As noticed by previous researchers [17, 19, 20], using an adaptive data structure one can provide a solution that dynamically adjusts its parameters based on actual conditions encountered and can be approximately as good as the best existing method for each specific access pattern.

Combining funnels allow the user to devise a general *adaptive strategy* that is optimized for only a few cases, and lets the processors modify the remaining parameters on the fly based on the actual load incurred. As an indicator of the load, funnels use the number of collisions a processor is involved in during each access to the object. Few

collisions serve as evidence of 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 of 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, at the cost of increased latency.

Since we know of no method that supports adding large numbers of processors without an increase in latency, our goal is to keep this increase as low as possible. Decisions on parameter changes are thus made locally by each processor: layers or funnels don't actually grow or shrink. Instead, each processor independently chooses its random layer location from a subrange of the full width, and starts traversal at a given layer from the full depth of layers available. Figure 9 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 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 (layer) 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.

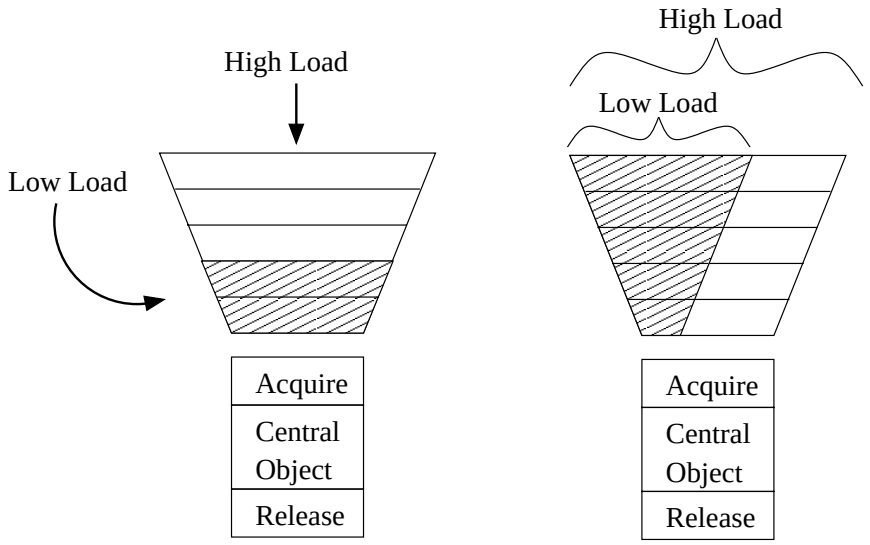


Figure 9: Two methods of adapting layer sizes to different loads. Shaded areas are used when the load is low.

3.3 Specific Adaptive Strategies

The following are our adaptive strategies for the fetch-and-add and stacks structures.

fetch-and-add 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. Processor p then 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.

stack For a stack 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 adapt to different machine loads with very little overhead. 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. While there are algorithms that support changing the size of a diffracting tree on the fly in response to load changes [7], they are substantially more complicated than our adaption strategies. 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. A particularly bad adaption scheme might cause processors to make wildly inaccurate layer size decisions lead to low performance, but correctness is never in jeopardy.

4 Performance

This section presents our performance benchmarks and the empirical evidence we collected using them. We begin with a general discussion and then present the specific performance results collected for the fetch-and-add and stack structures.

Research [14, 27, 29] has shown that in order to scale well, data structures must be parallel. Avoiding contention is not enough, and throughput must actually increase as more processors are added to the system. Currently, combining trees and elimination trees (detailed below) are the most effective parallel fetch-and-add and stack structures, respectively. We compared combining funnels to these algorithms. We also compared them 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 distributed-shared-memory multiprocessor similar to the MIT *Alewife* machine [2] of Agarwal et al. The simulation was conducted using the *Proteus*[4] multiprocessor simulator developed by Brewer et al. [4, 5]. The simulated Alewife machine is a 256 processor ccNUMA multiprocessor with realistic memory bandwidth and latency. We ran Proteus with accurate network simulation, which traces every packet and models contention and communication hot-spots. Though this is not a real 256 node machine, we note that previous research by Della-Libera [6] has shown that with appropriate scaling, Proteus simulates a 32 node Alewife machine[5] accurately for the kinds of data structures tested in this paper.

Proteus simulates parallel code by multiplexing several parallel threads on a single CPU. Each thread runs on its own virtual CPU with accompanying local memory, cache, and communications hardware, keeping track of how much time is spent using each component. In order to facilitate fast simulations, Proteus does not perform complete hardware simulations. Instead, operations which are local (do not interact with the parallel environment) are run directly on the simulating machine's CPU and memory. The amount of time used for local calculations is added to the time spent performing (simulated) globally visible operations to derive each thread's notion of the current time.

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

[5] Though Alewife was designed to scale to hundreds of processors, the largest machine currently available has 32 nodes.

Proteus makes sure a thread can only see global events within the scope of its local time. The machine we simulated had 256 processors in a mesh topology, each with 2^{13} bytes of memory and a cache of 256 lines of 32 bytes per line.

```
global count

FnA_benchmark()
{
   while(count < N) {
     w = random(0, work)
     for(i=0; i<w; i++)
       ;
     start = TIME()
     a = fetch_and_add(1)
     latency = TIME() - start
     count++
   }
}
```

```
global count

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

Figure 10: Code for benchmarking fetch-and-add (*left*) and stack (*right*) implementations. Global variables are seen by all threads but require no synchronization to access.

In our benchmarks, processors alternated between performing local work and accessing the shared object being tested, as detailed in Figure 10. Though these benchmarks are not real applications, they allowed us to accurately measure the difference in performance between our method and other algorithms. This approach has been used extensively in the literature [3, 7, 14, 16, 19, 22, 23, 27, 28, 29].

In order to choose the combining layer parameters (layer width, depth of funnel, delay times, etc.) we ran a series of preliminary tests aimed at finding the best parameters for the highest load, that is, 256 processors and no local work. We used these parameters in all our experiments. We ran two sets of benchmarks, 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 measured *latency*, the amount of time (in cycles) it takes for an average access to the object. Using the notation of Figure 10 this is the sum of all `latency` values divided by `count`.

In experiments where less than 256 processors were used, we nevertheless simulated a machine of 256 processors, though not all processors participated in running the algorithm. Processor 0 always participated, and other processors were added in order of increasing distance (on the mesh) from processor 0. We first added processors at distance 1 on the mesh from 0, then those at distance 2, and so on.

4.1 Fetch-and-Add Performance

The graph in the left part of Figure 11 shows the performance of the fetch-and-add implementations as the number of processors changes and where 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 [22]. The graph shows that combining funnels are substantially more expensive than the MCS-lock only 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. At 48 processors and beyond the latency is significantly worse than that of the other methods tested.

Combining funnels outperform optimal height combining trees by a small amount for all levels of concurrency. Notice, that this result indicates that adapting between different sized combining trees [19] would be slower than employing adaptive funnels. The performance of both methods is very close since both are trying to accomplish the same task. However, combining funnels can adjust to the actual number of processors present, whereas combining trees must be given this number explicitly. This is evidenced by the curve for constant height combining trees. Examination of the curve shows a significant gap between this method and combining funnels. When using 64 processors, the combining tree is two levels too deep and has 70% higher latency. Halve the number of processors, and the tree becomes three levels too deep and twice as slow as our method.

The right-hand graph of Figure 11 provides an even more compelling argument for the power of adaptive strategies. Using 256 processors, we must employ a combining tree of height eight even though, for higher local work loads, the tree is unlikely to reach that level of concurrency. The case where the number of processors is high – but not maximal – is the worst possible scenario for combining trees. A slight drop in concurrency immediately leads to a substantial decrease in the amount of combining. Processors that do not combine are essentially locked out of the path to the root. The result: a "spike" in the latency curve. As local work increases this effect slowly diminishes. Lower levels of concurrency increase each processor's chances of ascending to the root with little waiting.[6]

No 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. Unfortunately, under sparse access patterns, funnels are still up to three times slower than MCS-locks. Nevertheless, one should remember that MCS-locks are specifically tailored to the low concurrency case.

To summarize these results: the low overhead of MCS-locks makes them appealing for objects with low contention. In these cases combining funnels are not appropriate, though they are still substantially better then constant depth combining trees. For objects which are usually accessed by more than 16 processors concurrently, the extra parallelism afforded by combining funnels more than compensates for the extra overhead, and makes them the better choice.

4.2 Stack Performance

In [27], Shavit and Touitou compare different stack implementations and the one based on elimination trees is shown to outperform the rest. An elimination tree is a non-linearizable stack implementation shaped as a complete binary tree of l levels, where the root and all internal nodes are simple data structures called elimination-balancers. Processors traverse the tree from the root to the leaves by passing through the elimination-balancers. At each leaf a regular serial stack is located, protected by a lock, on which `PUSH` and `POP` operations can proceed normally. At the heart of each elimination-balancer is a one-bit variable called a toggle-bit with one or more prism arrays before it. When two processors collide in a prism they can either *eliminate*, if one is a `PUSH` operation and the other a `POP`, in which case the `PUSH`-ing processor delivers its element directly to the `POP`-ing processor and both exit the elimination-tree immediately. Or, if both

[6] A more detailed analysis of combining tree performance under various conditions in appears in [29].

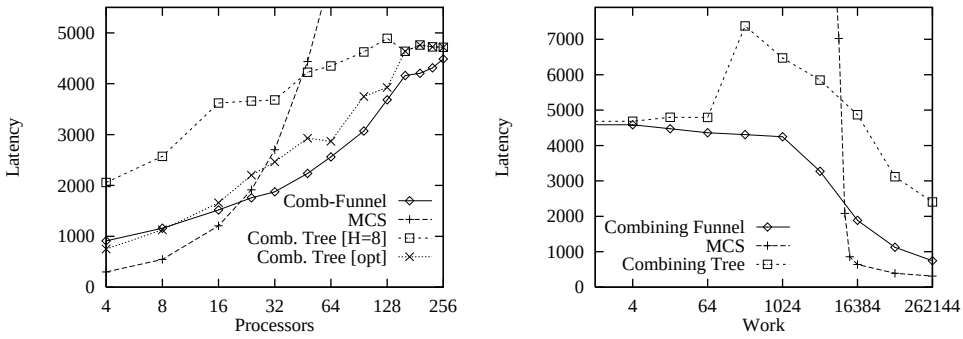


Figure 11: Latency of different fetch-and-add implementations with varying number of processors (*left*) and local work (*right*).

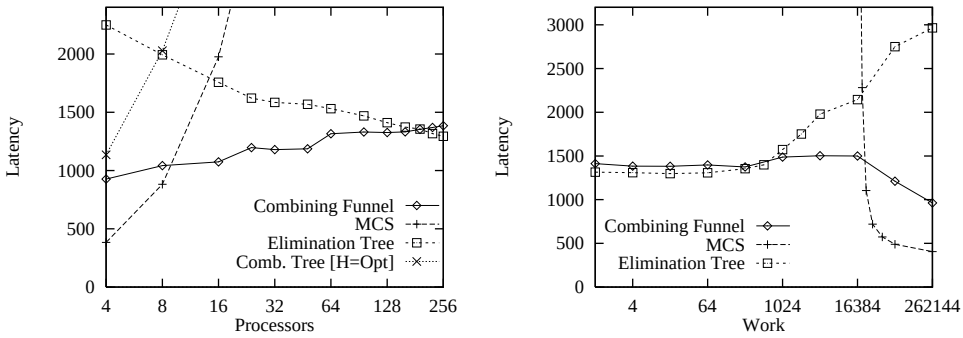


Figure 12: Latency of different stack implementations with varying number of processors (*left*) and local work (*right*).

processors are performing the same operation, *diffract*, one processor descends to the node's left child and the other to the right child. Only processors which do not collide in a funnel, and instead gain access to a balancer's toggle-bit, perform the equivalent of a fetch-and-complement operation on it. They then decide whether to descend to the left or right child based on the result of complement operation. Processors performing `PUSH` go left if the result was "0" and right on a "1", those performing `POP` do the opposite. Thus, any operation performed on the tree can end either part-way down if successfully eliminated, or, having passed through l elimination-balancers, end at one of the stacks at the leaves.

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. The way we turn a combining tree into a stack is very similar to our combining funnel method, except that it does not support elimination. Processors combine requests of the same kind going up the tree, when one reaches the root it increments (or decrements) the counter there by the total number of requests combined with. A distribution phase follows, in which every combined processor is given a location in the stack. To ensure linearizability we use the same ticket-lock/"go ahead" type method we used in combining funnels. We found that combining trees are always substantially slower than combining funnels, ten 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

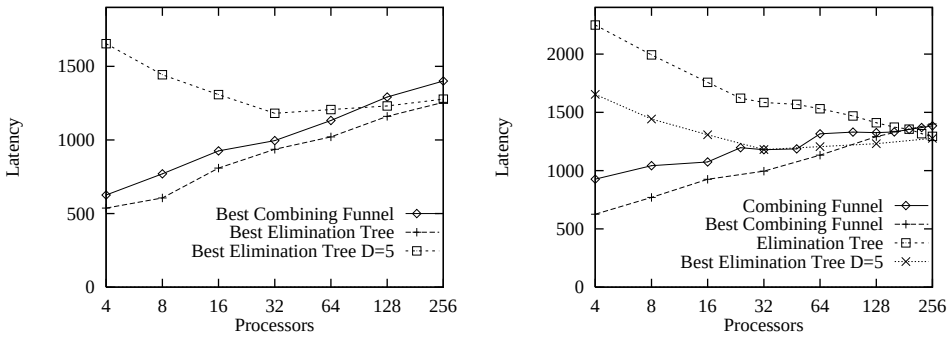


Figure 13: Latency of different stack implementations using best possible parameters for each concurrency level (*left*); and comparison of best parameters to those achieved by using an adaptive strategy (*right*).

these results in our graphs, concentrating on other methods instead.

In Figure 12 we again see that the latency of simple MCS based locking is unsurpassed at low concurrency levels. However, already at eight processors the difference between this method and ours is quite small. Adding more processors causes a rise in contention slowing this method considerably. These results are mirrored in the graph on the right, where MCS based stacks can be seen to underperform for all but the highest local work loads i.e. the lowest concurrency levels. At the opposite end of the spectrum we see that at 256 processors elimination trees outperform combining funnels by about 10% (recall that the parameters used by both methods have been optimized for 256 processors). 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 [27]), while the curve for combining funnels slopes up – fewer processors mean lower latency. Thus at 64 processors the difference 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[7] 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. It might seem that an appropriate adaption strategy applied to elimination trees would be able to combat this problem, but as we shall see this is not the case.

To study the behavior of our adaption strategy we conducted a series of experiments to hand-pick an optimum set of parameters for each level of concurrency. We then compared the performance of the adaptive strategy to the "best parameter set." These results are summarized in Figure 13. For elimination trees we differentiate between parameters which achieve lowest latency for a given depth tree, and those which also pick the optimum depth for the tree. We believe this is a reasonable distinction since changing the depth of the tree "on the fly" is much harder than altering other parameters. The left-hand graph of Figure 13 shows latency when using best parameters. Elimination trees with adjustable depth can be seen to consistently outperform combining funnels by about 10%. However, when depth is kept constant we see the familiar downwards curve of elimination tree latency. This means that even under the most favorable conditions this method cannot escape the consequences of reduced chances for diffraction. It further proves that any adaption method that does not allow for tree shrinkage cannot deliver

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

good performance for all concurrency levels. Comparing the performance of the best set of parameters to those achieved using adaption (on the right-hand side of Figure 13) we see that the adaptive strategy is less than 10% slower when the number of processors is 128 or more, 20% slower when there are between 16 and 128 processors, but below that the gap grows to 40%. Our adaption techniques appear to work reasonably well, though there is room for improvement here, especially in low load situations.

It is important to note that when we picked the combining funnel parameters for use in our tests, we optimized for one particular case: maximal load. Where maximal load is defined as all processors repeatedly accessing a single funnel. The adaptive strategy did all the rest. This is significant since it implies there is no need to re-optimize for each application, only for each size of machine. Thus, one set of parameters can be used for all applications running on a machine of 256 processors – the set which gives best performance at maximal load. Moreover, the graphs in Figures 13 serve as an indication of the sensitivity of the data structure to different choices of parameters. For example, they show that using parameters optimized for 256 processors on a machine with only 32, costs about 20% in performance. Whether this is acceptable or not is application dependent.

5 Discussion

We presented Combining funnels, a generalized framework for developing highly concurrent data objects. We hope to have convinced the reader that they offer a simple, structured, step-by-step approach that can be used to create effective parallel fetch-and-add and stack objects.

We note that a similar adaptive variation of combining, the *counting pyramid*, was introduced by Wattenhofer and Widmayer [32] concurrently and independently of the preliminary version of this paper [30]. In their paper the authors provide a queuing theory based analysis of the counting pyramid's performance. The counting pyramid differs from our work in several fundamental ways: (1) it is a message passing algorithm, (2) does not employ any form of elimination, and (3) its adaptivity is not in response to the load incurred by processors while traversing the tree (rather, processors choose a "level of entry" based on the frequency of their individual increment requests). However, the analysis of counting pyramids suggests that it would be interesting to apply similar techniques to evaluate the performance of combining funnels.

The kernels used in our benchmarks do not exhibit real world behavior. In a well designed application it is rare to find all processors hammering on a single object. However, our methods already show an advantage in performance when the number of processors is just 16. We believe that in applications running on hundreds of processors objects accessed by 16 processors concurrently might not be so rare. Similarly, as applications are scaled to thousands of processors, the ability to "plug in" data structures which can effectively handle hundreds of concurrent accesses will make programmers' lives much easier. They will be able to concentrate more on the algorithm and spend less time worrying about the contention on each individual object.

The stack described here is only one of many possible stack implementations that fit within the combining funnel framework. We chose this one since, of those we tried, it gave the best performance. Other possibilities include having colliding equal operations copy operands from one to the other so that processors carry arrays of operands rather than trees, this makes elimination very simple even for different sized arrays. Another is to employ linked-lists of operands, this way combining lists takes only $O(1)$ operations as does inserting lists into the stack. For both variations removing n values from the stack becomes $O(n)$, rather than $O(\log n)$ as in the method we employed. On our system these methods did not do as well, on others they might.

Similarly, the adaptive strategy employed in our implementation was chosen arbitrarily since it seemed to make sense and performed well in our tests. We have not carried out research aimed at determining the best possible adaption strategy, though clearly this is an interesting problem.

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.

The implementations given here are all done in a shared memory environment, however translation of the algorithms into message passing is straightforward and follows [14, 29]. The work of Herlihy et al. in [14] would seem to indicate that one could expect substantially better overall performance for the message passing versions, at least on Alewife. The results of [29] show that low bandwidth/high locality message passing systems favour algorithms with highly optimized static assignment. On such systems combining trees might outperform funnels. However, it is an interesting research question whether the ability to adapt or the predetermined optimized layout are the dominant factor in determining performance at lower concurrency levels.

Finally, we have recently been able to use combining funnels to implement highly scalable priority queues [31]. We believe many other funnel based data-structures have yet to be developed.

6 Acknowledgments

We would like to thank Dan Touitou for his many insightful suggestions and comments.

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. Reactive Diffracting Trees. Master's Thesis, Massachusetts Institute of Technology, 1997.

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

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

[9] 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.

[10] 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.

[11] 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.

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

[13] R. Gupta and C.R. Hill. A Scalable Implementation of Barrier Synchronization Using an Adaptive Combining Tree. *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pp. 54–63, 1989.

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

[15] 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.

[16] G.C. Hunt, M.M. Michael, S. Parthasarathy and M.L. Scott. An Efficient Algorithm for Concurrent Priority Queue Heaps. In *Information Processing Letters*, 60(3):151–157, November 1996.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] M.M. Michael and M.L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. Technical Report University of Rochester, Computer Science Department, TR600, Dec. 1995

[24] J.M. Mellor-Crummey and M.L. Scott. Fast, Contention-Free Combining Tree Barriers. *Intl. J. of Parallel Programming*, 22(4), August 1994.

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

[26] 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.

[27] 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.

[28] N. Shavit, E. Upfal and A. Zemach. A Steady State Analysis of Diffracting Trees. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996, pp. 33–41.

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

[30] N. Shavit and A. Zemach. Combining Funnels: A New Twist on an Old Tale... In *Proceedings of the 17th Annual ACM Symposium on Principals of Distributed Computing (PODC), Santa Barbara*, pages 61-70, August 1998.

[31] N. Shavit and A. Zemach. Scalable Concurrent Priority Queue Algorithms In *Proceedings of the 18th Annual ACM Symposium on Principals of Distributed Computing (PODC), Atlanta*, pages 113-122, May 1999.

[32] R. Wattenhofer and P. Widmayer The Counting Pyramid: an Adaptive Distributed Counting Scheme. *Proceedings of the 5th International Colloquium on Structural Information and Communication Complexity*, 1998.

[33] 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.

A The Data Structures Code

A.1 Fetch and Add

Figure 14 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. The 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 the code of a given processor. Lines 1–5 set up the data structures for an operation. Aside from setting up several counters, a processor initializes its `my` structure to reflect the fact that it is operating on `obj` (Line 4) with operation `op` (Line 5).

```
Fetch_and_Add( object *obj , operation_type *op )
{
 1    n = 0
 2    subtotal = op->sum
 3    op->result = EMPTY
 4    my->Location = obj                                  // I'm working on obj
 5    my->Operation = op                                  // I'm applying op
 6    while( 1 ) {
 7      for(i=0; i < NumberOflayers; i++) {               // foreach layer
 8        r = random() % obj->layerWidth[i]               // choose random location
 9        q = SWAP( obj->layer[i][r] , MYID )             // read q
10        if ( CaS(my->Location, obj, NULL ) ) {          // Lock myself
11            if ( CaS(q->Location, obj, NULL ) )  {      // Lock q
12               list[n++] = (q , q->Operation->sum )     // collision succeeds!
13               op->sum += q->Operation->sum             // add q and update op
14            }
15            my->Location = obj                          // unlock myself
16         }
17         else goto distribute                           // I've been collided with
18         for(i=0;i<obj->Spin[l]; i++)                   // delay loop
19           if ( my->Location != obj) goto distribute
20      }
21      if(CaS(my->Location,obj,NULL)) {                  // Lock myself
22        val = obj->counter;
23        if(CaS(obj->counter , val, val + op->sum))  {   // Update counter
24          op->result = val                              // update successful
25          goto distribute                               // distribute results
26        }
27        my->Location=obj                                // update failed, unlock
28      }
29      else goto distribute
30    }
31  distribute:
32    while( op->result == EMPTY) /* spin */;            // wait for parent's result
33    val = op->result
34    for(i=0; i<n; i++) {                                // distribute results
35      ( q , qsum ) = list[i]                            // to children
36      q->Operation->result = val + subtotal
37      subtotal += qsum
38    }
}
```

Figure 14: Code for Fetch and Add implementation

Next, the processor enters the funnel and repeats the following sequence of operations for each of the funnel layers. First, it picks a random location in the layer `r` (Line 8). It reads into `q` the ID of the processor written at `r` and writes its own ID in place (Line 9). Using a pair of compare-and-swap operations it accomplishs the "collision." The first compare-and-swap "locks" its `Location` pointer by replacing the object pointer with a `NULL`. The second compare-and-swap locks `q` in the same way. The success of both operations indicates that both processors are available for collision. A process then adds `q` to the list of its children, and updates its `Operation` field (Lines 12-13). It then resets its `Location` field to note that it allows more collisions to occur (Line 15). The `goto` in Line 17 is taken only if its attempt to lock its own `Location` failed, since then it follows that some other processor has collided with it, and it must wait for that processor to distribute a result. Finally, after attempting a collision, the processor delays at the current funnel layer to give another processor a chance to collide with it (Lines 18–19).

After passing through all funnel layers, the processor attempts to apply its combined operation to the shared object. First it must avoid races by preventing collisions during its attempt. This is again done through locking its `Location` pointer (Line 21). It now reads the current value of the counter `val`, and attempts to update it to the new value `val + sum` using a compare-and-swap operation. If the compare-and-swap succeeded, the counter has been updated and the processor can jump to the distribution phase (Line 25). Otherwise, it unlocks its `Location` pointer and traverses the funnel again (Lines 27,30). On a machine that supports an atomic fetch-and-add, it might be more efficient to apply the operation `val = fetch-and-add(obj->Counter, op->sum)` instead of using the compare-and-swap on Line 23. This would increment the counter directly and avoid the chance of the compare-and-swap failing.

Lines 32–38 implement the distribution phase. A processor first waits for a value to be provided by its parent (if it has no parent, `op->result` will be set on Line 24). It then iterates over its list of children and provides each one with a result by setting its `op->result` field (lines 34-37).

A.2 Stack

Code for the stack implementation is slightly more involved. Due to its length we have broken the code into two separate figures. Figure 15 shows how collisions occur and the central stack is handled, and Figure 16 deals with processors once they have collided. The same public data is used here as in the fetch-and-add algorithm with an additional per-processor public word `Comm` used to relay different types of information between processors.

A processor begins as before with a setup phase (lines 1–4) followed by an attempt to collide (lines 7–10). Notice that the `for` loop in Line 6 has an added exit condition. In the fetch-and-add implementation a processor advances one level at every attempt, so it is guaranteed to exit the funnel and attempt access to the central object after at most `obj->Levels` attempts. The stack implementation advances levels only after a successful collision. In order to avoid starvation when collisions are rare, a processor is provided a "back-door" to exit the funnel after every `obj->Attempts` collision attempts.

A processor then chooses `r` and reads `q` as before. It then tries to collide using two compare-and-swap operations (Lines 7-10). Here it uses not only the object's pointer in the `Location` array, but the current level as well. This means the compare-and-swap with `q` will only succeed if `q` is also operating on this object and `q` is currently on the same level as the process is. In this way collisions between processors on different levels are avoided by default.

When two processors collide, they can have either the same operation or opposing operations. In the case of both operations being equal (Line 11) the processor adds `q` to its list of children and advances to the next level. It also resets its attempts counter `n` (Line 14) to give it more chances to collide on the next layer. When the operations are

```
Stack( object *obj , operation_type *op )
{
 1    l = 0
 2    my->Comm = EMPTY
 3    my->Op = op
 4    my->Location = < obj , l >                      // setup location
 5    while(1) {
 6      for(n=0;n < obj->Attempts && l < obj->Levels; n++) {
 7        r = random() % obj->Width[l]                  // choose r
 8        q = SWAP(obj->layer[l][r], my)                // read q
 9        if(CaS(my->Location, <id,l>, NULL)) {         // attempt collision
10          if(CaS(q->Location, <id,l>, NULL)) {
11            if( q->Op->command == op->command) {      // success & equal
12              my->List[l] = q                         // add to list
13              my->Location = <id , ++l>               // go to next level
14              n = 0                                   // reset attempts
15            }
16            else {                                    // success & opposite
17              if(op->command == PUSH) {               //  pusher gets COLLIDE
18                 my->Comm = < COLLIDE, q, 0 >         //  and address of popper.
19              else                                    //  Popper not updated
20                 q->Comm = < COLLIDE , my, 0 >        //  and waits.
21              goto collided
22            }
23          }
24          else my->Location = < id , l>               // unlock myself
25        }
26        else goto collided
27        for(i=0;i<obj->Spin; i++)                     // delay loop
28          if( my->Location != <id, l> ) goto collided
29      }
30      if(CaS(my->Location, <id,l>, NULL)) {             // lock myself
31        if(Acquired(obj->lock)) {                       // try to lock SP
32          sp = obj->SP                                  //  success!
33          obj->SP = update_sp(sp, op->command , 1<<l)   //  update sp
34          myticket = obj->TICKET++                      //  get ticket for
35          Release(obj->lock)                            //  update phase
36          myplace = update_sp(sp , op->command , 1)
37          for(i=l-1; i>=0; i--) {                       // distribute stack
38              my->List[i]->Comm = <STACK , my, sp >     // locations to
39              sp = update_sp(sp, op->command , 1<<i)    // children
40          }
41          while (obj->NOWSERVING != myticket) ;         // wait my turn
42          my->Go = 1<<l                                 // everyone in my tree
43          ret = obj->do_single(myplace,op)              // can access stack now
44          Decrement(my->Go)
45          while(my->Go > 0)                             // when everyone done
46          obj->NOWSERVING ++                            // next proc goes
47          return ret
48        }
49        else my->Location = < id , l >
50      }
51      else goto collided
52    }
```

Figure 15: Part 1 of code for stack implementation

```
53  collided:
54    while( my->Comm == EMPTY) ;                    // wait for status update
55    < type , q , val > = my->Comm
56    switch( type ) {
57     case STACK:                                   // root acquired SP
58       sp = update_sp(val, op->command, 1)         //  distribute a stack
59       for(i=l-1; i>=0; i--) {                     //  location to each child
60         my->List[i]->Comm = < STACK , q, sp >
61         sp = update_sp(sp, op->command, 1<<i)
62       }
63       while(q->Go==0) ;                           // wait for go-ahead
64       op->result = obj->do_single(val,op)         // update stack
65       Decrement(q->Go)
66       break
67     case COLLIDE                                  // I'm an eliminated pusher
68       for(i=l-1; i>=0; i--)                       // each child gets a popper
69         my->list[i]->Comm = < COLLIDE , q->list[i], 0 >
70       q->Comm = < VALUE , 0, op->data >           // send my popper my value
71       break
72     case VALUE                                    // I'm an eliminated popper
73       op->result = val                            // return value got from pusher
74    }
}
```

Figure 16: Part 2 of the code for stack implementation

opposing an elimination occurs. The processor performing the `PUSH` is given a pointer to the other processor's operation, in its `Comm` word (Lines 17-20). This allows the pushing processor to write the values in its tree to the processors waiting to perform a `POP`.

As before, Line 24 resets the `Location` pointer after an unsuccessful collision attempt. Line 26's `goto` is taken if a processor has been collided with. Lines 27–28 delay a processor at its current layer to allow more collisions.

Lines 30–51 detail the operations performed on the central stack. A processor begins by locking its own `Location` (Line 30) and attempting to acquire the lock on the stack-pointer (Line 31). If the first locking attempt fails, it was collided with (Line 51). If the second fails, it indicates load on the stack and the processor unlocks its `Location` and returns to the funnel. Having acquired the lock on the stack-pointer the processor can now safely update it (Line 34). 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 the appropriate bounds. The size of the update in line 33 is 2^l since, having been on the l-th layer, a processor is known to be at the root of a tree of 2^l operations. While still holding the lock on the stack-pointer the processor gets a ticket for the the next phase – updating the stack itself (Line 35). Ticket in hand, it releases the lock on the stack-pointer (Line 36). At this point it holds no locks (though it has a placed reserved on the ticket line). It thus takes the time to distribute stack locations to it children. Lines 37–40 implement a distribution phase similar to the one used in the fetch-and-add implementation, and provide each child with a unique location in the stack where they can apply their `PUSH` or `POP` operation.

When a processor's turn arrives at the ticket lock, it gives all its children a "go-ahead" signal by setting its own "go" field (Line 42). It then performs its single operation on the stack and waits for processors in its tree to complete their operations. The loop in Line 45 serves as a barrier and prevents passing the ticket to the next waiting processor before all operations in this tree are done.

Any processor that collided jumps to Line 53. Note that when a collision occurs it is not yet known if a processor's traversal of the funnel will end in an elimination or an update of the central stack. Thus, the code in lines 54–74 must be prepared to handle both possibilities. The `type` field written to the `Comm` word informs the processor of the fate of the tree. `STACK` indicates that it is operating on the central stack, and `val` is its location. First it distributes locations to its children (Lines 58–62), and then it waits for the root's "go-ahead" (Line 63). When the go ahead is received the processor updates the stack and decrements the barrier (Line 65).

The identifier `COLLIDE` indicates an elimination. When a processor's operation is a `PUSH`, i.e. it must push its value to a partner performing a `POP`, a pointer to the partner `q` is provided to the processor in the `Comm` word by its parent. The same is done for the processor's children, by iterating over `q`'s list of children, and writing one pointer in each of its children's `Comm` word (Lines 68–69). Finally, the processor updates `q`'s `Comm` word with the value it is pushing and the identifier `VALUE` (Line 70). A processor that sees the `VALUE` identifier knows that the value it must return was written in the third field of the `Comm` word (Line 73).