Fusible Data Structures for Fault-Tolerance

Vijay K. Garg* and Vinit Ogale
Parallel and Distributed Systems Laboratory
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084, USA
(garg,ogale@ece.utexas.edu)

Keywords: fault-tolerance, replication, data structures.

Abstract

We introduce the concept of *fusible data structures* to maintain fault-tolerant data in distributed programs. Given a fusible data structure it is possible to combine a set of such structures into a single fused structure that is smaller than the combined size of the original structures. When any of the original data structures is updated, the fused structure can be updated incrementally using local information about the update and does not need to be entirely recomputed. In case of a failure, the fused structure, along with the correct original data structures, can be used to efficiently reconstruct the failed structure. We show that many commonly used data structures like arrays, hash tables, stacks and queues are fusible and we present algorithms to fuse and recover such structures. This approach often requires significantly less space than conventional backups by replication and still allows efficient operations on the original data structures. For example, our experiments with fault servers on a distributed system suggests that for a system with k servers, this approach requires k times less space than the active replication approach.

*supported in part by the NSF Grants CNS-0509024, Texas Education Board Grant 781, SRC Grant 2006-TJ-1426, and Cullen Trust for Higher Education Endowed Professorship

1 Introduction

Fault-tolerance of servers is a fundamental and important topic in distributed and parallel systems. Active replication is a standard method to achieve fault-tolerance in face of various failures in a distributed system. It is almost considered a self-evident truth that, to tolerate crash of t servers, one must have $t + 1$ copies of identical processes. This approach, for example, is the underlying assumption in the work on replicated state machine approach [5, 11, 13, 17, 15, 16, 3]. In that work, if all $t + 1$ state machines (or servers) start with the identical state, are completely deterministic in execution, and agree on the set and the order of commands they execute, then they will have the identical state at all times. This means that failure of t of them will leave at least one copy available. The optimality of this approach has generally not been questioned. In this paper, we initiate study of fusible data structures that allow practical techniques for fault-tolerance with lower space and communication overhead than the replicated state machine approach.

In data storage and communication, coding theory is extensively used to recover from faults. For example, RAID disks use disk striping and parity based schemes (or erasure codes) to recover from the disk faults [10, 2, 12]. As another example, network coding [8, 1] has been used for recovering from packet loss or to reduce communication overhead for multi-cast. In these applications, the data is viewed as a set of data entities such as disk blocks for storage applications and packets for network applications. By using coding theory techniques [7] one can get much better space utilization than, for example, simple replication. The reader is referred to the tutorial by Plank for an overview of the commonly used Reed-Solomon codes [12]. All these techniques are oblivious to the structure of the data. The details of actual operations on the data are ignored and the coding techniques simply recompute the encoded backups after any write update. Returning to our earlier example of tolerating crash failures for servers, one can view the memory of the server as a set of pages and apply coding theory to maintain code words. This approach, however, many not be practical because a small change in data may require recomputation of the backup for one or more pages. This results in a high computational and communication overhead.

In this paper we introduce the concept of *fusible data structures* that enable us to efficiently maintain fault tolerant data in parallel or distributed programs. Our technique is based around

the actual structure of the data and the operations used to change the data. We exploit our extra knowledge of the data structure and the permitted operations to reduce the space and communications overhead and, at the same time, allowing incremental updates to the data.

Fusible data structures satisfy three main properties: recovery, space constraint and efficient maintenance. The recovery property ensures that in case of a failure, the fused structure, along with the remaining original data structures, can be used to reconstruct the failed structure. The space constraint ensures that the number of nodes in the fused structures is strictly smaller than the number of nodes in the original structures. Finally, the efficient maintenance property ensures that when any of the original data structures is updated, the fused structure can be updated incrementally using local information about the update and does not need to be entirely recomputed.

We show that many commonly used data structures like arrays, hash tables, stacks and queues are fusible and present efficient algorithms to fuse such structures. As a concrete example, we consider a lock server in a distributed system that maintains and coordinates use of a lock. The lock server maintains the record of the process that has the queue and the list of all pending requests. Assume that the size of the pending request queue is n_{max}. Traditionally, if fault-tolerance from a crash is required, we would keep two copies of the queue. If there are k such lock servers in the system, and each one is replicated, we get the space overhead of kn_{max}. In our proposal, we keep a single back-up queue for all k servers that is obtained by fusing the original queues. Our fused queue uses $O(n_{max})$ space, supports recovery and can be updated efficiently when any of the primary queue gets updated. This technique results in k-fold savings.

Another example is a server that maintains a hash table (with linked-list buckets for collisions) of size N. Active replication for tolerating a single crash will result in doubling of the space overhead. In our scheme, we divide up the hash table into k smaller hash tables, each of size $O(N/k)$. In addition, we keep a fused table of size $O(N/k)$ that is computed from the k original hash tables. With $k+1$ machines, we can tolerate a single crash with space overhead of $O(N/k)$ instead of $O(N)$ used by the replication scheme. In fact, the simple replication corresponds to the degenerate case when k equals 1. In this example, note that our scheme requires availability of more processors, $k+1$ instead of the two processes required in the traditional methods. In many

Number of processes	10	20	30	40	50
Queues	8.6	16.4	23.9	31.7	38.7
Stacks	9.6	18.3	26.7	34.8	43.7
Priority Queues	1.3	1.6	1.8	2.0	2.2
Sets	6.5	12.3	18.5	24.3	30.5
Linked Lists	1.5	1.8	2.1	2.4	2.6

Table 1: Experimental results: (space required by replication)/(space required by fusion)

cases this disadvantage is offset by the advantages in space reduction offered by fusion. Obviously, as in the case of the queue, the fused hash structure can also be to backup separate hash tables on different servers in a distributed system.

We have experimentally evaluated our technique by implementing a library supporting arrays, queues, stacks, sets, priority queues, linked lists and hash tables in a distributed programming framework. A brief summary of our experimental results is given in table 1. The space requirements are averaged over multiple runs. For example, the space required by active replication of 50 queues is around 40 times more than the space required by fusion. For systems with k servers, queues, arrays, stacks and sets require around $k/2$ times less space than replication. The improvement is not so drastic for linked lists and priority queues, but the space requirements are still reduced by a factor of two or more when k is large.

In this paper, we focus primarily on single crash failures. The ideas in this paper can be easily generalized to the case when there can be t concurrent failures by using erasure codes (like Reed-Solomon codes [7]). We briefly discuss this in section 6.

In summary, this paper makes the following contributions:

- We introduce the notion of fusible data structures that provide fault-tolerance with reduced space overhead as compared to active replication based schemes that are widely used now.
- We present algorithms to efficiently fuse standard data structures including arrays, stacks, queues, sets, priority queues and hash tables.
- We have a Java implementation of fusible data structures which can be transparently used in distributed programs. Since the overhead is minimal, this can also be used to easily

increase reliability in programs which inherently do not have any fault tolerance. For example, instead of using the standard queue, the programmer can easily use the queue from our implementation library and get fault tolerance without significant programming effort or computational overhead.

- We apply our technique to an extensively studied problem of lock server in distributed systems. Our experimental results show k-fold improvement in space complexity for maintaining k lock servers in a fault-tolerant manner.

The remainder of the paper is organized as follows: we first define and explain fusible data structures. We then present algorithms to fuse commonly used data structures including arrays, stacks, queues, lists and tables. Section 5 discusses the performance of fusible techniques in practical applications. We then compare existing approaches using erasure codes or replication with the fusion approach discussed in this paper.

2 Fusible Data Structures

A data structure is a tuple consisting of a set of nodes and some *auxiliary information*. The auxiliary information may be implicit or explicit and delineates the 'structure' of the data nodes. For example, every element of an array is a node and the auxiliary information that is implicit in the definition of the array specifies that the nodes are contiguous and in a specific order. On the other hand, a linked list has explicit auxiliary data consisting of the next and previous node pointers for each node along with the head and tail pointers of the list.

Every data structure has a set of valid operations associated with it. For example, `push` and `pop` are the valid operations on a stack. The data structures may also have read-only operations, but we ignore them in this paper since they do not affect the backup data structure.

Given k instances of a data structure, our objective is to efficiently maintain a more efficient backup of these structures than active replication. If a failure occurs, it should be possible to reconstruct any structure on the failed process using the backup and the remaining structures. We assume that failures are restricted to crash failures. The recovered data structure needs to have

the exact values of the nodes and *equivalent* auxiliary data. For example, the actual values of the pointers in a linked list will not be preserved and the reconstruction will simply be a list with identical values in the same order.

We define a *fusible* data structure X as follows.

Definition 1 *Let $x_1, \ldots, x_k$ be instances of a data structure X where $k > 1$ and each node in X has size s. Assume that each of $x_1, \ldots, x_k$ contain $n_1, \ldots, n_k$ nodes respectively and let $N = \Sigma_{i=1}^{k} n_i$. Then X is fusible if there exists a data structure Y with an instance y such that:*

1. **(Recovery)** *Given any $k-1$ of $x_1, \ldots, x_k$ and y, there exists an algorithm to recreate the missing x_i.*

2. **(Space Constraint)** *The number of nodes in y is strictly less than N. The size of any node in Y is $O(k+s)$ and the space required for any additional data maintained by Y is independent of N.*

3. **(Decentralized Maintenance)** *For any operation that updates one of the x_i, there exists an operation to update y using information only from y and x_i. The time required for the operation on y is of the same order as the operation of x with respect to the number of nodes in x_i and y.*

The data structure Y is called a *fusion* of X. We refer to the process of computing y as *fusing* the structures $x_1, \ldots, x_k$ and to the instance y as the fused structure or fusion.

The *recovery* property is the crucial property required for fault-tolerance. Whenever one of the x_i is unavailable (for example, due to a process failure), it should be possible to recreate x_i with the remaining objects and y. When the server that stores y crashes, then our requirement is slightly weaker. The recovered object may even have different structure; the only requirement is that it be a valid fusion of x_i's.

The *space constraint* property rules out trivial algorithms based on simple replication. Note that, simple replication satisfies recovery property; we can easily recover from one fault, but it does not satisfy the space constraint. On the other hand, erasure coding is data structure oblivious and though it results in space savings, it does not allow efficient update of data.

The *decentralized maintenance* property ensures that as any object x_i changes, there is a way to update y without involving objects other than x_i and y. The cost of maintenance is one of the main metrics for comparison with standard erasure coding methods. For example, assume that we need to maintain fusion of linked lists. Consider the strategy of maintaining y as simple xor of bit arrays corresponding to the memory pages that contain x_i's. Any change in x_i, say inserting a new node, requires the re-computation of y with time complexity that may depend on the size of x_i and y. With fusible data structures, we exploit properties of the data structure so that the cost of updating y is of the same order as of the cost of updating x_i.

An obvious fusible data structure is a single bit. Assume that the operations defined on the bit are: `get` and `set`, where `get` returns the value of the bit and `set` sets the bit to the provided value. Let $x_1, \ldots, x_k$ be k bits. The fused data structure is also a bit initialized to xor of all x_i. Whenever any operation $set(b)$ is issued on x_i, y is updated as :

$$y := y \otimes x_i \otimes b$$

The fused bit satisfies our recovery property because the missing bit can be obtained by xoring the remaining bits. It satisfies the space reduction property because the sum of sizes of all objects is k bits, whereas the fused bit uses a single bit. Finally, the maintenance of y requires value only from x_i and y.

The single bit example can be easily generalized to any data structure with a fixed number of bits like char, int, float etc.

In the previous example, the `set` operation requires information from x_i to update y. This is not always necessary and we could have data structures with operations that do not require any input from the original structure to update the fused structure.

Definition 2 *(Independent operation) An operation is independent with respect to a fusible data structure X, if its corresponding operation on the fusion Y does not require any information from X*

If all the operations on a data structure are independent the structure is said to be independent.

The fused bit data structure described previously is not independent because the `set` operation requires the value of x_i to update its fusion y. Now consider the single bit data structure permitting the operations `get` and `toggle` with usual semantics. In this case, the maintenance operation for y is even simpler. Whenever any x_i is toggled, y is toggled as well. Thus no information from X is required to update Y. Hence this data structure is *independent*.

One more example of a fusible data structure is a counter with n bits that takes values from 0 to $2^n - 1$. The operations on the data structure are `set`, `increment` and `decrement`. Assume that both increment and decrement operations are modulo n. Given k counters one can keep the fused counter as *xor* of all the primary counters. This way the space overhead is n bits (instead of kn bits required by replication). However, none of the operations are independent. When a primary counter is updated, updating the fused counter requires the previous value of the primary counter. A better fusion method for the fused counter is to keep the modulo 2^n sum of the k primary counters . Given any $k-1$ primary counters and the fused counter that has sum modulo 2^n, one can easily derive the value of the missing counter. Again, we have the same space overhead of n bits. However, now the `increment` and the `decrement` operations can be performed on the fused data structure *independently*. The `set` operation is not independent. Note that, when the operations in the counter are simply `read` and `set`, then keeping the sum would not have any advantage over xor. The efficiency of fusible data structure crucially depends upon the operations.

In the next few sections we explore how standard computer science data structures such as arrays, stacks, queues, linked lists etc. can be fused.

3 Array based data structures

In the following discussion we use X to denote the original or the *source* data structure and Y to denote the fused structure. We assume that there are k instances, $x_1, \ldots, x_k$, of the source data structure and their fusion is denoted by y. The number of nodes in each of $x_1, \ldots, x_k$ is given by $n_1, \ldots, n_k$ respectively while N is the total number of source nodes, i.e., $N = \Sigma_{i=1}^{k} n_i$. The number of nodes in the fused structure is denoted by N_y. We often assume that the original data structures are nonempty to avoid adding boundary conditions to the algorithm. This assumption is

not integral to the algorithms and serves no other purpose apart from simplifying the presentation.

We first extend our example of a single bit to another simple fusible data structure: a bit array of size n. Assume that the following operations are allowed on an object of this class: *get(j)* that returns j^{th} bit and $set(j, b)$ that sets j^{th} bit to b. We now show,

Lemma 1 *The bit array data structure is fusible.*

Proof: In a bit array structure each bit constitutes a *node*. We maintain a fusion of k such arrays $x_1, \ldots, x_k$ by keeping another bit array y. The i^{th} bit of y is computed by xoring the i^{th} bits from each of $x_1, \ldots, x_k$. This takes care of the efficient update property. If the sizes of the arrays $x_1, \ldots, x_k$ are different then we take the largest of them as the size of y and pad the smaller arrays with zeros. To recover any bit array, it is sufficient to compute xor of the remaining bit arrays with y. The space constraint is satisfied since the number of nodes in y is $max(n_1, \ldots, n_k)$ and this is always less than N.

■

When the source arrays are of different sizes, we also need to maintain the sizes of the arrays in the fused structures, so that in case of a failure, the correctly sized array is recovered. It is not necessary to store k distinct sizes in the fused structure and a single value that is the xor of these values is stored.

We now generalize the bit array example.

Theorem 1 *If a data structure is fusible, then any fixed size array of that data structure is also fusible.*

Proof: We consider each element of the array to be a node. A valid fusion is another identical array, where the i^{th} element is a fusion of all the i^{th} nodes from each of the arrays. This fusion exists since it is given that the original data structure is fusible.

■

Thus, arrays of basic data types like short, int, float, double are fusible as well.

3.1 Array based Stacks and Queues

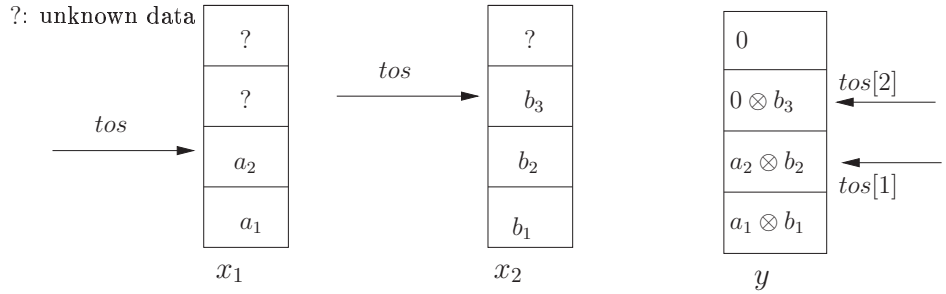


Figure 1: Stacks

In the rest of the paper we will use the $\otimes$ operator to denote fusing nodes in data structures. The actual nodes could be complex fusible objects and in such cases instead of bitwise xoring, we mean computing the fusion of those nodes. We now consider data structures that encapsulate some additional data besides the array and support different operations. The array based stack data structure maintains an array of data, an index `tos` pointing to the element in the array representing the top of the stack and the usual `push` and `pop` operations.

Lemma 2 *The array based stack data structure is fusible given $O(k)$ additional storage. The* `push` *operation is independent.*

Proof: We assume that all stacks are initially empty. The fused stack consists of the fusion of the arrays from the source stacks. We keep all the stack pointers at y individually. This additional $O(k)$ additional storage is required for the efficient maintenance property. The following push and pop operations satisfy the efficient maintenance property.

```
function x_i.push(newItem)
  x_i.array[x_i.tos] := newItem;
  x_i.tos++;
  y.push(i,newItem);
end function

function y.push(i, newItem)
  y.array[y.tos[i]] := y.array[y.tos[i]] ⊗ newItem;
  y.tos[i]++;
end function
```

push

```
function x_i.pop()
  x.tos[i] −−;
  y.pop(i, x_i.array[x_i.tos]);
return x_i.array[x_i.tos]

function y.pop(i, oldItem)
  y.tos[i] −−;
  y.array[y.tos[i]] := y.array[y.tos[i]] ⊗ oldItem;
end function
```

pop

When an element is pushed onto one of the source stacks, x_i, the source stack is updated as usual and the request is forwarded to the fused stack. The fused stack does not require any additional information from x_i, i.e., the `push` operation is independent. During a `pop` operation, we xor the corresponding value in y with the value that would be returned by $x_i.pop()$.

The number of elements, n_y, in the array corresponding to the fused stack is the maximum of $n_1, \ldots, n_k$ which is less than N. Therefore, the space constraint is satisfied.

```
function y.recover(failedProcess)
/*Assuming that all source stacks have the same size*/
  recoveredArray := new Array[y.array.size];
  for j = 0 to tos[failedProcess] − 1
      recItem := y[j];
      foreach process p ≠ failedProcess
          if (j < tos[p]) recItem := recItem ⊗ x_p.array[j];
      recoveredArray[j] := recItem;
return recoveredArray, tos[failedProcess]
```

From the algorithm, it is obvious that any stack $x_{failedProc}$ can be recovered by simply xoring the corresponding elements of the other original stacks with the fused stack.

■

Note that, in the usual implementation of stacks, it is not required that the popped entry be zeroed. However, for fusible data structures it is essential to clear the element during a `pop` operation to ensure that the next `push` operates correctly.

It is easy to accommodate different sized stacks in the fused data structure. In this case, similar to the arrays example in lemma 2, the fusion y contains an array that is as large as the biggest stack.

Circular array based queues can be implemented similarly by zeroing out deleted elements and keeping the individual head and tail pointers.

Lemma 3 *The circular array based queue data structure is fusible.*

4 Dynamic data structures: Stacks, Queues, Linked Lists

So far, we have looked at data structures based on arrays. We now look structures like stacks, queues and sets based on linked lists. Instead of using a generic fusion algorithm for all structures based on linked lists, we use specific properties of each structure to make the fusion more efficient wherever possible.

Stacks are lists that allow insertions and deletions from the same end. Queues allow insertions at one end and deletions from the other. Dequeues are a generalization of stacks and queues that allow insertions and deletions from either end. The implementation of the fusion of these three structures is similar.

To allow arbitrary deletions or insertions, i.e., for a generic linked list we use a different fused structure. Priority queues and sets use the general linked list algorithm. A priority queue allows arbitrary insertions and deletions at a single end. A set permits arbitrary deletions, but since the order of elements is not important the algorithm can insert all new nodes at the best possible location, say one of the ends.

4.1 Stacks

In section 3.1, we introduced an algorithm for the fusion of an array based stack structure. We now look at the linked list based stacks, i.e., a linked list which supports inserts and deletes at only one end, say the tail. The fused stack is basically another linked list based stack that contains k tail pointers, one for each contributing stack x_i.

When an element $newItem$ is pushed onto stack x_i, then

- if $tail[i]$ is the last element of the fused stack, i.e, $tail[i].next = null$, a new element is inserted at the end of the fused queue and $tail[i]$ is updated.
- otherwise, $newItem$ is xored with $tail[i].next$ and $tail[i]$ is set to $tail[i].next$.

```
function y.push(i, newItem)
  tail[i] := tail[i].next;
  if(tail[i] = null)
      tail[i] := new node();
  tail[i].value := tail[i].next.value ⊗ newItem;
  tail[i] := tail[i].next;
end function
```

push

```
function y.pop(i, oldItem)
  tail[i].value := tail[i].value ⊗ oldItem;
  oldTail := tail[i];
  tail[i] := tail[i].previous;
  if((oldTail.next = null) ∧(∀ j: tail[j] ≠ oldTail))
      delete oldTail;
end function
```

pop

When a node is popped from a stack x_i, the value of that node is read from x_i and passed on to the fused stack. In the fused stack, the node pointed to by $tail[i]$ is xored with the old value. If $tail[i]$ is the last node in the fused list and no other $tail[j]$ points to $tail[i]$, then the node corresponding to $tail[i]$ can be safely deleted once the value of $tail[i]$ is updated. Note that in this case, a push takes O(1) time but a pop operation may require $O(k)$ time, since we check if any other tail points to the node being deleted. This satisfies the efficient maintenance property of fusible structures since the time required is independent of the size of the total number of nodes in the original data structure. For constant time pop operations, the algorithm for fused queues in section 4.2.1 can be applied for stacks.

The fusion of the list based stack requires no more nodes than the maximum number of nodes in any of the source stacks. The size of each node in the fused stack is the same as s, the size of the nodes in the original stack X. The only extra space overhead is the k tail pointers maintained. If all the stacks are approximately of the same size, the space required is k times less than the space required by active replication.

4.2 Queues

The fusion of list based queues is more involved than stacks since both ends of the list are dynamic. We assume that elements in a queue are inserted at the tail of any queue and only the head of a

queue can be deleted.

We begin with examining why an algorithm similar to the list based stacks cannot be used for queues. If we modify the algorithm so that it applies to queues, we get a structure that seems to be a fused queue but does not always satisfy the space constraint since it could have exactly N nodes in the worst case. An example of when this could happen is shown in figure 2.

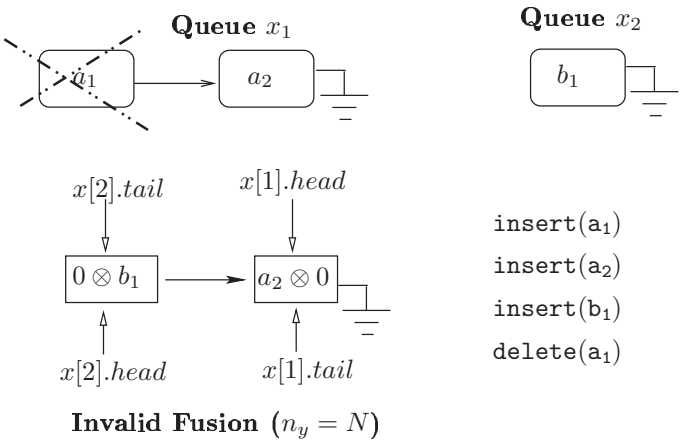


Figure 2: The resultant data structure is not a valid fusion

To ensure that it meets the space constraint, we need to merge nodes $head[i]$ and $head[i].previous$ if possible after $head[i]$ is deleted in the fused structure. Determining if the nodes can be merged in the data structure described above can take $O(N)$ time, violating the efficient update property.

We now present an algorithm for fusing queues that satisfies the space constraint and the efficient update property by maintaining an extra $log(k)$ bits at each node.

4.2.1 Fused Queues

As in the previous algorithm, the fused data structure is implemented as a linked list with each node in the list containing the *xor* of elements from the original queues. Each node in the fused structure also contains an extra variable, the *reference count* which is a count of the number of source queues whose data is maintained by that node. The reference count enables us to decide when a node in the fused structure can be safely deleted (when its reference count is 0) or merged. The fused data structure Y contains a list of head and tail pointers for each component queue, pointing to the corresponding node in fused list. For example, in figure 3(ii), $x[1].tail$ has a reference count of 1

since it contains a value from only x_1 while $x[1].head$ contains the fusion of values from x_1 and x_2 and hence has a count of 2.

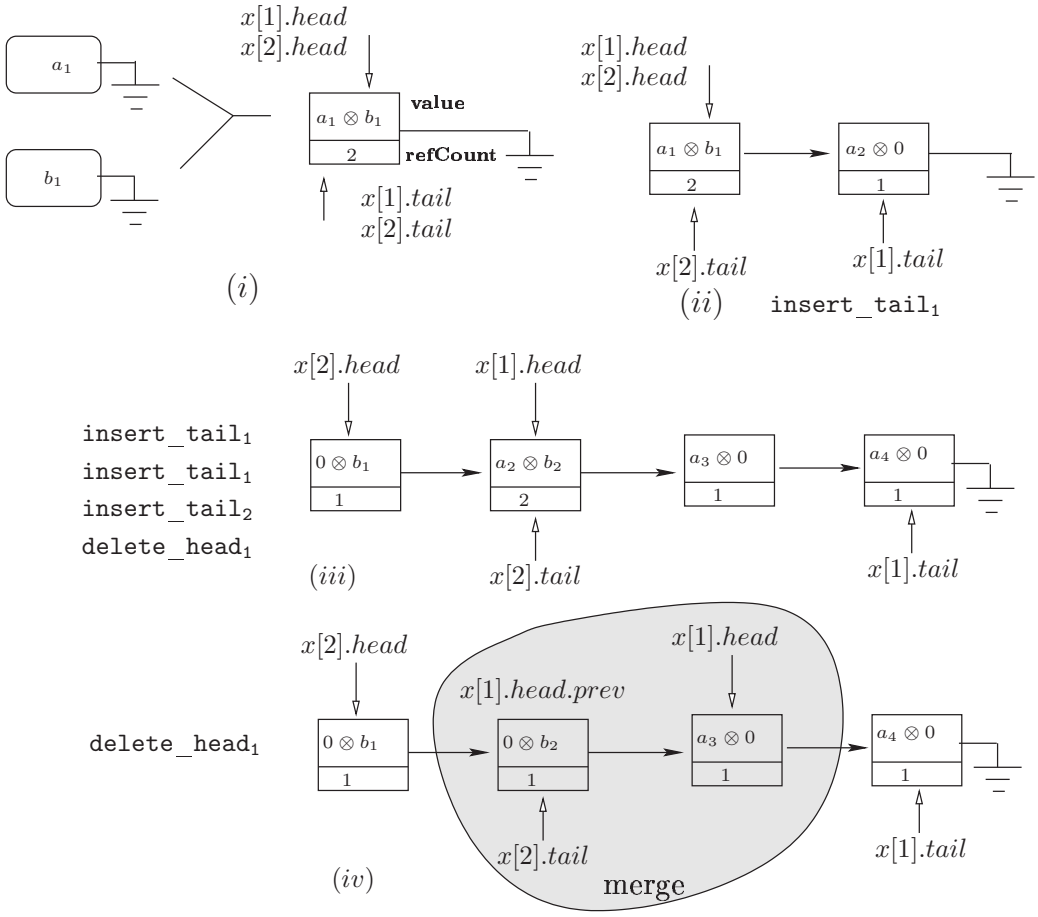


Figure 3: Example of a fused queue

```
function y.insertTail(i, newItem)
  if (tail[i].next ≠ null)
      tail[i] := tail[i].next;
      tail[i].value := tail[i].value ⊗ newItem;
  else tail[i] := new_node(newItem);
  tail[i].refCount ++ ;
end function
```

insert

```
function y.deleteHead(i, oldItem)
  oldHead := head[i];
  oldHead.value := oldHead.value ⊗ oldItem;
  oldHead.referenceCount −−;
  head[i] := oldHead.next;
  if(oldHead.refCount = 0) delete(oldHead);
  if (head[i].refCount = 1 ∧
      head[i].previous.refCount = 1)
      merge(head[i], head[i].previous);
end function
```

delete

As in the case of stacks, deleting a value from the fused structure requires access to the old value that is to be deleted. The old value is xored with the value at $head[i]$, $head[i]$ is set to $head[i].next$ and the reference count is decremented by one. If the new reference count is 0 then old node at $head[i]$ is deleted. The reference count also simplifies the merging. The new $head[i]$ and $head[i].previous$ are merged when both their reference counts are equal to one.

1. Figure 3(i) shows a fusion of two queues, x_1 and x_2 containing one element each. In this case, the fused structure contains exactly one node containing the xor of a_1 and b_1. The pointers $x[1].head$, $x[1].tail$, $x[2].head$ and $x[2].tail$ point to this node and the reference count of this node is two.

2. The insertion of a new element is similar to pushing an element in the stack. Figure 3(ii) shows the fused structure after an element is inserted in queue x_1. A new node is created in the fused linked list and $x[i].tail$ is updated so that it points to this node. The reference count of the new node is one.

3. Figure 3(iii) shows the fused queue after a sequence of two inserts and a delete on queue x_1 and an insert in queue x_2.

4. Now if the head of queue x_1 is deleted, the resulting fused queue is shown in figure 3(iv). In this case, the reference count of both $x[1].head$ and $x[1].head.previous$ is one. Hence each of these nodes contains data from a distinct source queue and the nodes are merged together forming a single node by xoring the values and setting the reference count to two. Note that $x[2].tail$ and $x[1].head$ both point to the same newly merged node.

Lemma 4 *The number of nodes in the fused queue is less than N.*

Proof: (outline) Since we delete empty nodes the maximum number of nodes in the fused queue is bounded by N. N nodes in the fused queue implies that there are adjacent nodes with a reference count of one containing data from different processes (for example the scenario in figure 2). However, this can never happen with the algorithm just described since adjacent nodes are always merged after a delete whenever each of them contains elements from a different process.

■

Hence the space constraint is satisfied. Note that, unlike the list based stack algorithm discussed earlier, this algorithm requires $O(1)$ time for insertions as well as deletions. It requires an extra $log(k)$ bits at every node of the fused list.

4.3 Dequeues

Dequeues are a generalization of list based stacks and queues. The reference count based fusion implementation of linked lists is easily extensible to a dequeue allowing two new operations `insert_head` and `delete_tail`.

4.4 Linked lists

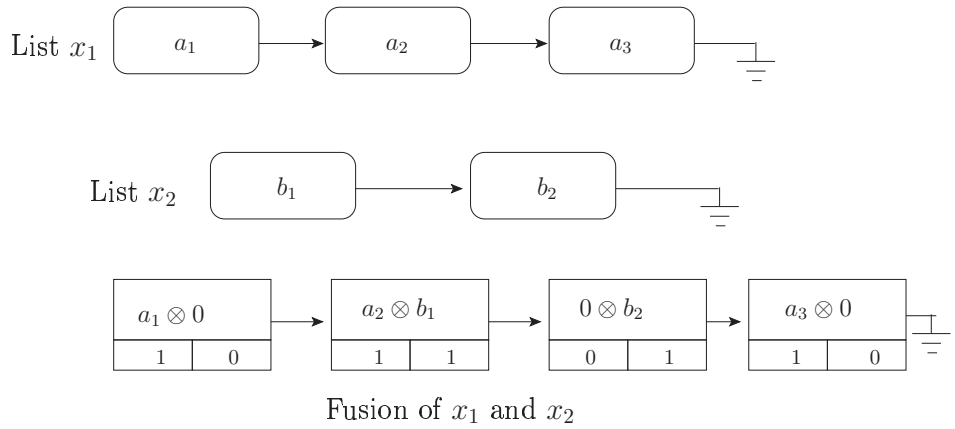


Figure 4: Linked Lists

We now examine a generic linked list. The fused structure for a linked list is also valid for priority queues and sets. Linked lists can be fused together in a manner similar to queues, except that we also maintain a bit array of size k, at each node of the fused list, with a bit for every source linked list. Each bit in the array enables us to determine if a node contains data from the list corresponding to that bit. It also enables us to determine if adjacent nodes can be merged.

It is not necessary to maintain a list of head or tail pointers corresponding to each source list in this case. However in priority queues or sets, the head or tail pointers can optionally be maintained to eliminate the search overhead for deletes and inserts.

```
structure nodeInFusedList
   pointers next, prev
   boolean bitField[1 ... numLists]
   value
end structure
```

To insert or delete an element, the source list x_i sends the element along with its index in the source list. The algorithm examines the bit field of each fused node, counting the number of nodes which have data from x_i to determines the location for the insert or delete. To insert a node after say the m^{th} element of the source list, the algorithm advances through the fused list till it encounters m^{th} elements with their i^{th} value in the bitfield set to nonempty. If the next node does not have an empty space for source list i, a new node with the input value is created. In some cases there may be multiple successive vacant spots and depending on the application it might be desirable to insert the new element in the middle space instead of the first.

```
function y.insert(i, index, newItem)
   node = y.head;
   while(count ≤ index)
        if(node.bitField[i]) count ++;
        node := node.next;
   if (node.next = null ∧ node.next.bitField[i] = notEmpty)
             node := insert_new_node_after(node);
        else node = node.next;
   node.value := node.value ⊗ newItem;
   node.bitField[i] := notEmpty; /* notEmpty = 1 */
end function
```

insert

```
function y.delete(i, index, oldItem)
   node = y.head
   while (count < index)
        if(node.bitField[i]) count ++ ;
        node := node.next;
   node.value := node.value ⊗ oldItem;
   node.bitField[i] := empty; /*empty = 0*/
   mergeNodes(node);
end function
```

delete

4.4.1 Performance Considerations

Though the number of nodes in a fused linked list is guaranteed to be less than the number of nodes in the source lists, the fused structure has an overhead of k bits at each node. Hence in the adversarial case, the actual space required by the fusion algorithm could be more than the space required by a replication scheme.

If N_y is the number of nodes in the fused structure the space required by the fused list is $N_y(s+k)$. Fusion is advantageous when $(s+k).N_y < s.N$, i.e., $N/N_y > (s+k)/s$. Note that N_y

is always less than N. When $s >> k$, $N/N_y > (s+k)/s$ is always satisfied and the fused structure outperforms active replication. If we assume that s and k are approximately equal then the N/N_y needs to be greater than two. Note that the actual value of N/N_y may vary from run to run. In our simulations, N/N_y was always around two or more when k was greater than ten. Thus it is reasonable to assume that the fused linked list outperforms replication unless s is significantly less than k, that is, when we have very small nodes in the linked list and a large number of source lists.

Sets are special cases of linked lists in which the order of nodes does not matter. Hence we always insert new nodes at one end of the list. In our simulations, this drastically improved the performance of the fused structure and the ratio N_y/N always hovered close to the theoretical maximal value k. Therefore in practice, our fusion algorithm can always be applied to sets. Priority queues, which allow insertions at any locations and deletions at one end, perform similar to normal linked lists and the results for the general linked list are applicable to priority queues too.

4.5 Hash Tables

A chained hash table can be constructed using a fixed sized array of the sets. Sets are a special case of linked lists and a the fusion of a set can be computed as described in subsection 4.4. Such a table is fusible and the fusion consists an array of fused linked lists from the individual hash tables.

Lemma 5 *Hash tables are fusible.*

Proof: Sets are a special case of linked lists and linked lists are fusible. A hash table is a fixed size array of such lists. Therefore, from theorem 1, it follows that hash tables are fusible.

■

5 Experimental Evaluation

For some structures like stacks and simple arrays, it is clear from the algorithms that fusion always performs much better than replication. However for queues and other structures, in the worst case, only one node may be fused, resulting in space requirements that are almost equivalent to active

replication. In the optimal scenario, however, the number of nodes required by the fused structure is k times smaller than active replication.

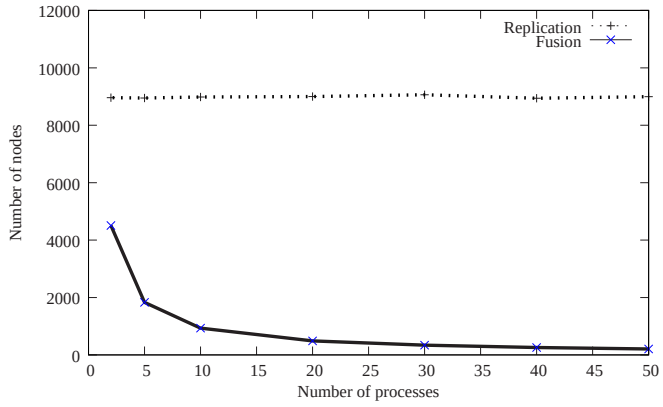
We have implemented a library of common data structures for regular use based on the distributed programming framework used in [4]. Using this, we examined the performance of the data structures in different scenarios. We also implemented a k-server mutual exclusion distributed application.

5.1 Fault-Tolerant Lock Algorithm

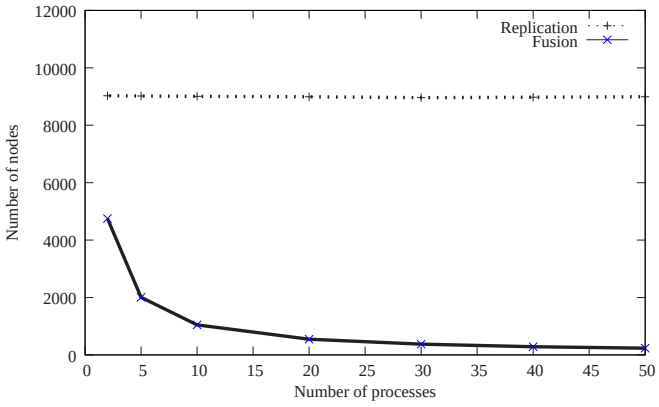
We now look at a distributed computing application that uses queues using fusion for backup. It consists of k lock servers that arbitrate access to shared resources. Each lock server maintains a queue of pending requests from clients. We use a single fused queue to backup the queues in all k servers. Every time a server modifies its queue the changes are propagated to the fused queue which is updated as described before.

We modified the lock server program [4] by simply substituting fusible queues instead of the normal queues. The backup space required by k servers was drastically lower, by a factor of more than $k/2$, as compared to active replication.

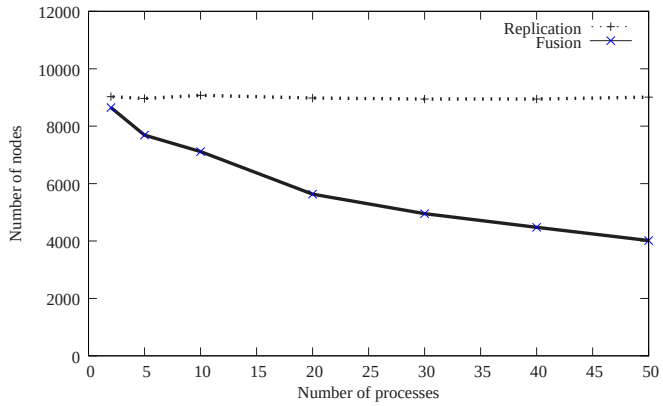
5.2 Simulation results



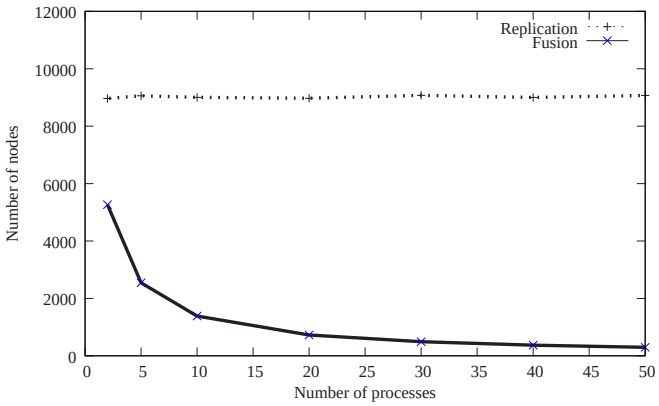
(Stacks)



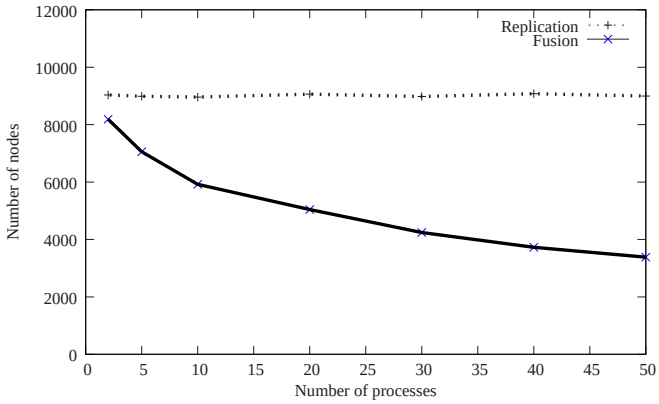
(Queues)



(Priority Queues)



(Sets)



(Linked Lists)

As seen in the lock server scenario, queues perform well in almost all scenarios resulting in a savings factor of $k/2$ or more. We also tested queues and other data structures by simulating random inserts and deletes and averaging the results over multiple runs. To examine the performance when the data structures are large, we biased the simulations to favor inserts over deletes.

Like queues, sets and stacks showed noticeable and consistent savings of around $k/2$. General linked lists and priority queues however do not show consistently significant savings, requiring around half the number of nodes as compared to replication. The graphs compare the number of nodes required by the queues with the number of nodes required by active replication. As we can see, stacks, queues and sets show a huge improvement (in the order of k) over replication. Fusion

of priority queues and linked lists requires around half the space required by replication.

6 Comparison with Error Correcting Codes and Replication

We now briefly discuss the relation between fusion, error correcting codes and replication.

6.1 Error Correcting Codes

It might initially seem that the concept of nodes in fusion is very similar to pages or blocks in erasure codes. We would like to emphasize that this is not true, since our techniques revolves around maintaining and determining the *structure* of the nodes in the original data structure. For example, to backup a linked lists using error codes, the exact nodes in the linked list could be used as pages of the erasure code. However, inserting nodes or deleting them would not make sense in such a scenario without using techniques similar to those described in this paper. The crux is that error correcting codes are data oblivious while fusion inherently depends on the structure of the data and the operations permitted on it.

The fusion technique introduced in this paper can be loosely divided into the following parts.

1. Distinguishing between structural information and node data in a data structure.
2. Identifying and maintaining a fused structure that allows us to save storage space by fusing together nodes.
3. Techniques to fuse individual node data (throughout this paper we have use bitwise xoring or modulo addition for this).
4. Updating the backup structure efficiently when operations on the source structures modify some data.

Error correcting codes deal exclusively with the third item on the list, .i.e., fusing a set of data blocks into one or more backup blocks allowing recovery in case of failures. We have used xoring which is a parity based erasure coding, assuming only one failure. Our approach is directly extensible to the case of t failures using t backup structures by using other erasure coding techniques

(like Reed-Solomon codes [7]) to fuse each node together. The algorithms presented in this paper deal with steps 1, 2 and 4 and remain unchanged. Thus, fusion is orthogonal to error coding and advances in error codes can be transparently included in the fusion techniques.

6.2 Comparison with Replication

We have seen that the algorithms for fusing data structures, result in better space utilization than replication. However, recovery of faulty processes is much more efficient with replication. Hence, when failures are frequent, using replication might be preferable to fusion techniques.

Also the operations on the some of the fused data structures require more time than replication. For example linked list operations are slower by a factor of k. This is generally not a matter of concern since the slowdown affects only the backup process and can never be too large due to the efficient update property. However, there could be scenarios where operations on the backup server need to be as fast as possible and replication would be a better choice in such cases.

State machine replication is easily extensible for byzantine faults by increasing the number of backups [6, 14]. The fusion techniques discussed in this paper, in contrast, are not directly applicable for byzantine errors.

7 Conclusion

In this paper, we question the optimality of replication approaches used for server backups and examine the structure and operations on data to develop fused data structures. Our experiments with the lock server application indicate that as expected, fused data structures work much better than replication. We have implemented a library with commonly used data structures in Java [9]. In distributed applications, it is very easy to substitute standard data structures with the data structures from our library, thus allowing programs to be fault tolerant without any significant overhead.

This paper presents algorithms for the fusion of some common data structures. An interesting avenue for research would be to develop fusion techniques for other application specific data structures. Also, the fused structure for linked lists described in this paper does not perform as well as

queues or sets. It would be interesting to explore new fusion algorithms for linked lists.

References

[1] Byers, J. W., Luby, M., Mitzenmacher, M., and Rege, A. A digital fountain approach to reliable distribution of bulk data. *SIGCOMM Comput. Commun. Rev. 28*, 4 (1998), 56–67.

[2] Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A. Raid: high-performance, reliable secondary storage. *ACM Comput. Surv. 26*, 2 (1994), 145–185.

[3] Gafni, E., and Lamport, L. Disk paxos. *Distrib. Comput. 16*, 1 (2003), 1–20.

[4] Garg, V. K. *Concurrent and Distributed Computing in Java*. Wiley & Sons, 2004.

[5] Lamport, L. The implementation of reliable distributed multiprocess systems. *Computer networks 2* (1978), 95–114.

[6] Lamport, L., Shostak, R., and Pease, M. The byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4*, 3 (1982), 382–401.

[7] Lint, J. H. V. *Introduction to Coding Theory*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[8] Luby, M. G., Mitzenmacher, M., Shokrollahi, M. A., Spielman, D. A., and Stemann, V. Practical loss-resilient codes. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing* (New York, NY, USA, 1997), ACM Press, pp. 150–159.

[9] Ogale, V. A., and Garg, V. K. Fusible data structure library. In *Parallel and Distributed Systems Laboratory, http://maple.ece.utexas.edu/software* (2006).

[10] Patterson, D. A., Gibson, G., and Katz, R. H. A case for redundant arrays of inexpensive disks (raid). In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1988), ACM Press, pp. 109–116.

[11] Pease, M., Shostak, R., and Lamport, L. Reaching agreement in the presence of faults. *J. ACM 27*, 2 (1980), 228–234.

[12] Plank, J. S. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience 27*, 9 (September 1997), 995–1012.

[13] Schneider, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys 22*, 4 (1990), 299–319.

[14] Schneider, F. B., and Zhou, L. Implementing trustworthy services using replicated state machines. *IEEE Security and Privacy 3*, 5 (2005), 34–43.

[15] Sivasubramanian, S., Szymaniak, M., Pierre, G., and van Steen, M. Replication for web hosting systems. *ACM Comput. Surv. 36*, 3 (2004), 291–334.

[16] Sussman, J. B., and Marzullo, K. Comparing primary-backup and state machines for crash failures. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1996), ACM Press, p. 90.

[17] Tenzakhti, F., Day, K., and Ould-Khaoua, M. Replication algorithms for the world-wide web. *J. Syst. Archit. 50*, 10 (2004), 591–605.