A Fusion-based Approach for Handling Multiple Faults in Distributed Systems

Bharath Balasubramanian, Vijay K. Garg
Parallel and Distributed Systems Laboratory,
Dept. of Electrical and Computer Engineering,
The University of Texas at Austin.
{balasubr,garg}@ece.utexas.edu

***Abstract*—The paper describes a technique to correct faults in large data structures hosted on distributed servers, based on the concept of fused backups. The prevalent solution to this problem is replication. Given n distinct data structures, replication requires nf additional replicas to correct f crash faults or $\lfloor f/2 \rfloor$ Byzantine faults among the data structures. If each of the primaries contains $O(m)$ nodes of $O(s)$ size each, this translates to $O(nmsf)$ total backup space. Our technique uses a combination of error/erasure correcting codes and selective replication to correct f crash faults (or $\lfloor f/2 \rfloor$ Byzantine faults) using just f additional backups consuming $O(msf)$ total backup space, while incurring minimal overhead during normal operation. Since the data is maintained in the coded form, recovery is costly as compared to replication. However, in a system with infrequent faults, the savings in space outweighs the cost of recovery. We explore the theory and algorithms for these fused backups and provide a Java implementation of fused backups for all the data structures in the Java 6 Collection Framework. Our experimental evaluation confirms that fused backups are space-efficient as compared to replication (almost n times), while they cause very little overhead for updates. Many real world distributed systems such as Google's map reduce framework or Amazon's distributed data store use replication to achieve reliability. An alternate, fusion-based design can result in significant savings in space as well as resources.**

I. Introduction

Distributed systems are often modeled as a set of independent servers interacting with clients through the use of messages. To efficiently store and manipulate data, these servers typically maintain large instances of data structures such as linked lists, queues and hash tables. Faults that occur in these servers are classified into two categories: crash faults [24] and Byzantine faults [12]. In the case of crash faults, the servers crash, leading to a loss in state of the data structures. In the case of Byzantine faults, the servers can reflect any arbitrary state of the data structures. *Active replication* [10], [18], [25], [27] is the prevalent solution to this problem. To correct f crash faults among n data structures, replication-based solutions maintain f backup copies of each primary, resulting in a total of nf backups. These copies can correct $\lfloor f/2 \rfloor$ Byzantine faults, since greater than $\lfloor f/2 \rfloor$ data structures are truthful. A common example is a set of lock servers that maintain and coordinate the use of locks. Such a server maintains a list of pending requests in the form of a queue. To correct three crash faults among, say five such queues, replication requires three backup copies of each queue, resulting in a total of fifteen backup queues. Though recovery is cheap and simple, for large values of n, this is expensive in terms of the space consumed by the backups.

Coding theory [2], [13], [19] is used as a space-efficient alternative to replication, both in the fields of communication and data storage. Data that needs to be transmitted across a channel is encoded using redundancy bits that can correct errors introduced by a noisy channel [26]. Applications of coding theory in the storage domain include RAID disks [16], [5] for persistent storage, network coding approaches for reducing losses in multi-cast [14], [3] or information dispersal algorithms (IDA) for fault tolerance in a set of data blocks [22], [4]. These solutions are oblivious to the structure of the underlying data and are rarely applied to backup active structures in main memory. In the example of the lock servers, in order to correct faults among the queues, a simple coding-theoretic solution will encode the memory block occupied by the lock server. Since the lock server is never maintained contiguously in main memory, a structure-oblivious solution will have to encode all memory blocks that are associated with the implementation of this lock server in main memory. This is not space efficient, since there could be a huge number of such blocks in the form of free lists, memory book keeping information etc. Also, every small change to the memory map associated with this lock has to be communicated to the backup, rendering it expensive in terms of communication and computation.

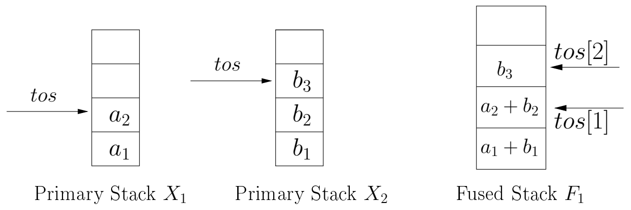


Fig. 1. Fault Tolerant Stacks

In this paper, we present a technique referred to as *fusion* which combines the best of both these worlds to achieve the space efficiency of coding and the minimal update overhead of replication. Given a set of primary data structures, we maintain a set of *fused* backup data structures that can correct f crash faults (or $\lfloor f/2 \rfloor$ Byzantine faults) among the primaries. These fused backups maintain the primary elements in the coded form to save space, while they replicate the index structure of each primary to enable efficient updates. Unlike coding-theoretic solutions, since the backups are designed at the abstraction of the data structure rather than the raw data behind them, we need not track the way data is maintained in memory. In figure 1, we show the fused backup corresponding to two primary array-based stacks X_1 and X_2. The backup is implemented as a stack whose nodes contain the sum of the values of the nodes in the primaries. We replicate the index structure of the primaries (just the top of stack pointers) at the fused stack. When an element a_3 is added to X_1, this element is sent to the fused stack and the value of the third node is updated to $a_3 + b_3$. In case of a delete, say b_3, the third node is updated to a_3. These set of data structures can correct one crash fault. For example, if X_1 crashes, the values of its nodes can be computed by subtracting the values of the nodes in X_2 from the appropriate nodes of F_1. We make the observation that in large practical systems, the size of data far exceeds the size of the index structure. Hence replicating the index structure at the fused backups is of insignificant size overhead. The real savings in space is achieved by fusing the values in the data nodes. Henceforth, for convenience, we just focus on crash faults. The extension to detection and correction of Byzantine faults is presented in section III-C.

Fusion is extremely space efficient while recovery is very cheap in replication. In figure 1, to correct one crash fault among X_1 and X_2, replication requires a backup copy for both X_1 and X_2, resulting in two backups containing five nodes in total as compared to the fusion-based solution that requires just one back containing three nodes. However, in case of a crash fault, recovery in replication just involves obtaining the value from the corresponding replica. Fusion needs a centralised recovery algorithm that obtains all available data structures and decodes the data nodes of the backups. In general, this is the key trade-off between replication and fusion. In systems with infrequent faults, the cost of recovery is an acceptable compromise for the savings in space achieved by fusion. Previous work on this topic [9] provides the algorithms to generate a single fused backup for array or list-based primaries, that can correct one crash fault. In Table I, we present the main differences among three backup solutions: replication, the older version of fusion, and the newer version of fusion presented in this paper. The following are the main contributions of this paper:

1) *Generic Design*: We extend the array and list-based fused backups of [9] to present a generic design for most commonly used data structures such as stacks, vectors, binary search trees, hash maps, hash tables etc. While [9] supports only add and remove operations on the primaries, we support both these operations and all other operations whose updates at the backups does not require decoding the values. For example, when a primary binary search tree is balanced, the update can be performed at the fused backups without any information from the primary.
2) *f-Fault Tolerance*: Using error/erasure correcting codes, we extend the xor/addition based 1-fault tolerant design of [9] to present f-fault tolerant data structures. In example 1, we can maintain another fused stack F_2 that has identical structure to F_1, but with nodes that contain the difference in values of the primary elements rather than the sum. The algorithms for updates are identical at the backups and each primary update is applied on both of them. These set of data structures can correct two crash faults. We extend this using Reed Solomon (RS) erasure codes [23], which are widely used to generate the optimal number of parity blocks in RAID-like systems. Using RS codes, we correct f crash faults among the primaries using just f additional fused backups.
3) *Space Optimality*: Given n primaries, each containing $O(m)$ nodes of size $O(s)$ each, the space complexity of a single backup for list-based primaries in [9] is $O(nms)$. This is as bad as the space required to maintain n replicas to correct one crash fault. For the design in this paper, the space occupied by a fused backup is $O(ms)$. To correct f faults, we require just $O(msf)$ backup space, achieving $O(n)$ times savings as compared to replication. We show that this is the minimum amount of space required to correct f crash faults.
4) *Update Efficiency*: In [9], the time taken to update the fused backup for linked lists is proportional to the number of nodes in the fused backup i.e. $O(nm)$. We show that the time complexity to update our fused backups is identical to that at the corresponding primary. In the case of linked lists, this is $O(m)$. Further, we show that by locking just a constant number of nodes, multiple primary threads can update the fused backups concurrently. Since the primaries are independent of each other, this could achieve significant speed-up.
5) *Order Independence*: The state of the fused backup in [9] is dependent on the order in which updates are received from the primaries. Hence, if we simply extend their algorithms for f-fault tolerance, then we need to ensure that all the backups receive updates from the primary in the same order. This implies the need for synchrony, which will cause considerable overhead during normal operation. In this paper, we show that as long as the updates from a single primary are received in FIFO order, the state of the fused backup is independent of the order of updates. FIFO order among primary updates is a strict requirement even for replication and can be easily implemented using TCP channels.
6) *Extension to Limited Backup Servers*: In practical systems, sufficient servers may not be available to host all the backup structures and hence, some of the backups have to be distributed among the servers hosting the primaries. These servers can crash, resulting in the loss of all data structures residing on them. Given a set of n data structures, each residing on a distinct

server, we prove that $\lceil n/(n+a-f) \rceil \cdot f$ backups are necessary and sufficient to correct f crash faults among the host servers, when there are only a additional servers available to host the backup structures.

7) *Real World Example, Amazon's Dynamo*: We apply the design of fused backups to a real world system and illustrate its practical usefulness. We consider Amazon's highly available key-value store: *Dynamo* [6], which is the data-store underlying many of the services exposed by Amazon to the end-user. Examples include the service that maintains shopping cart information or the one that maintains user state. Dynamo achieves its twin goals of fault tolerance (durability) and fast response time for writes (availability) using a simple replication-based approach. We propose an alternate design using a combination of both fused backups and replicas, which consumes far less space, while providing almost the same levels of durability, and availability for writes. We show that for a typical host cluster, where there are 100 dynamo hosts, the original approach requires 300 backup hosts, while our approach requires only 120 backup hosts. This translates to significant savings in both the space occupied by the hosts as well as the infrastructure costs such as power and resources consumed by them.

8) *Implementation and Results*: We provide a Java implementation of fused backups [1] using RS codes for all the data structures in the Java 6 Collection Framework. This covers most commonly used data structures such as sorted lists, stacks, vectors, hash maps, hash tables, tree map etc. We evaluate the performance of the fused backups presented in this paper with the one in [9] and replication. We consider three main parameters: backup space, update time at the backups and recovery time. The current version of fusion is very space efficient as compared to both replication (almost n times) and the older version (almost $n/2$ times). The time taken to update the backups is almost as much as replication (around 1.25 times slower) while it is much better than the older version (3 times faster). Recovery is much cheaper in replication (order of hundred times) but the current version of fusion performs almost $n/2$ times better than the older version. These results confirm the fact that the fused backups presented in this paper are space efficient while incurring very little overhead during normal operation.

TABLE I
FUSION VS. REPLICATION (n PRIMARIES CONTAINING $O(m)$ NODES OF SIZE $O(s)$, f FAULTS)

	Replication	**Old Fusion**	**New Fusion**
Types of Primaries	All	Arrays, Lists	All
Faults Corrected	$f \geq 1$	$f = 1$	$f \geq 1$
Number of Backups	nf	f	f
Backup Space	$O(nmsf)$	$O(nmsf)$	$O(msf)$
Update Optimality	Yes	No	Yes
Order Independence	Yes	No	Yes
Concurrent Updates	Yes	No	Yes
Recovery Time	$O(msf)$	$O(msf^2n^2)$	$O(msf^2n)$

II. MODEL AND NOTATION

Our system consists of independent distributed servers hosting data structures. We denote the n primary data structures, each residing on a distinct host, $X_1 \ldots X_n$. The backup data structures that are generated based on the idea of combining primary data are referred to as *fused backups* or *fused data structures*. The t fused backups, each residing on a distinct host are denoted $F_1 \ldots F_t$. The operator used to combine primary data is called the *fusion operator*. In figure 1, X_1, X_2 are the primaries, F_1 is the fused backup and the fusion operator is addition. We assume that the size of data far exceeds the overhead of maintaining the index structure at the backup. This is a reasonable assumption to make, since in most real world systems, the data is in the order of megabytes while the auxiliary structure is in the order of bytes (like next pointers in linked lists).

The updates to the servers in our system originate from a set of clients. When an update is sent to a primary, the data structure hosted on it is modified and the primary sends sufficient information to update the backups. We assume FIFO channels with reliable message delivery between the primaries and the backups. The updates to the backups are asynchronous and can be received in any order. The only requirement is that when there are faults, all the data structures in the system have acted on all the updates before the failed data structures are recovered.

The data structures in the system, both primaries and backups, may undergo crash or Byzantine faults. When a fault occurs, no updates are sent by the clients until the state of all the failed data structures have been recovered. Crash faults result in the loss of the current state of the data structure. We assume that the system in consideration can detect such faults. In the case of Byzantine faults, the data structures can be in any arbitrary state. We provide algorithms for the detection and correction of such faults. For recovery, we assume the presence of a trusted, recovery agent that can obtain all the available data structures, detect and correct faults.

III. FUSION-BASED FAULT TOLERANT DATA STRUCTURES

In [9], the authors present fusible data structures for array and list-based primaries. In this section, we present a generic design of fused backups for most commonly used data structures such as lists, stacks, vectors, trees, hash tables, maps etc.

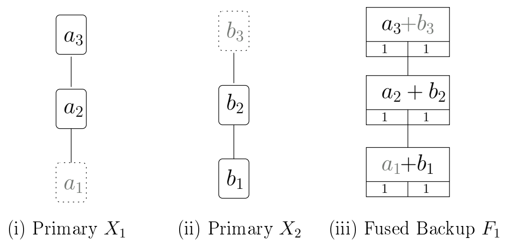


Fig. 2. Old Fusion [9]

Design Motivation: In [9], the authors present a design to fuse primary linked lists to correct one crash fault. The fused structure is a linked list whose nodes contain the *xor* of the primary values. Each node contains a bit array of size n with each bit indicating the presence of a primary element in that node. A primary is element inserted in the correct position at the backup by iterating through the fused nodes using the bit array and a similar operation is performed for deletes. An example is shown in figure 2 with two primaries and one backup. After the delete of primary elements a_1 and b_3 (shown in dotted lines), the first and third nodes of the fused backup are updated to b_1 and a_3 respectively (deleted elements in grey scale). After the deletes, while the primaries each contain only two nodes, the fused backup contains three nodes. If there are a series of inserts to the head of X_1 and to the tail of X_2 following this, the number of nodes in the fused backup will be very high. This brings us to the main design motivation of this section: Can we come up with a generic design for fused backups, for all types of data structures such that the fused backup contains only as many nodes as the largest primary (in this e.g. two nodes), while guaranteeing that updates are efficient? We present a solution for linked lists and then generalize it for complex data structures.

A. Fused Backups for Linked Lists

We use a combination of replication and erasure codes to implement fused backups each of which are identical in structure and differ only in the values of the data nodes. In our design of the fused backup, we maintain a stack of nodes, referred to as *fused nodes* that contains the data elements of the primaries in the coded form. The fused nodes at the same position across the backups contain the same primary elements and correspond to the code words of those elements. Figure 3 shows two primary linked lists X_1 and X_2 and two fused backups F_1 and F_2 that can correct two faults among the primaries. The fused node in the 0^{th} position at the backups contain the elements a_1 and b_1 with F_1 holding their sum and F_2 their difference. Along with the stack, at each fused backup, we also maintain auxiliary structures that replicate the index information of the primaries. The auxiliary structure corresponding to primary X_i at the fused backup is identical in structure to X_i, but while X_i consists data nodes, the auxiliary structure only contains pointers to the fused nodes. In the case of linked list based primaries, the auxiliary structures are simply linked lists. The savings in space are achieved because primary nodes are being fused, while updates are efficient since we maintain the ”structure” of each primary at the backup.

Overview: We begin with a high-level description on how we restrict the number of nodes in the backup stack. Elements belonging to primary X_i are simply inserted one following the other in the backup stack with a corresponding update to the index structure of X_i at the backup to preserve the actual ordering information. The case of deletes is more complex. If we just delete the element at the backup, then like in the case of figure 2, a hole will be created and the fused backups can grow very large. In our solution, we shift the top-most element of X_i in the backup stack, to plug this hole. This ensures that the stack never contains more nodes than the largest primary. Since the final element is present in the fused form, the primary has to send this value with every delete to enable this shift. To know which element to send with every delete, the primary has to track the order of its elements at the backup stack. We achieve this by maintaining an auxiliary list at the primary, which mimics the operations of the backup stack. When an element is inserted into the primary, we insert a pointer to this element at the end of its auxiliary list. When an element is deleted from the primary, we delete the element in the auxiliary list that contains a pointer to this element and shift the final auxiliary element to this position. Hence, the primary knows exactly which element to send with every delete. Figure 3 illustrates these operations with an example. We explain them in greater detail in the following paragraphs.

Inserts: Figure 4 shows the algorithms for the insert of a key-value pair at the primaries and the backups. At each primary X_i, along with the primary data structure we maintain an auxiliary list that mimics the operation of the backup stack. When the client sends an insert, if the key is not already present, the primary creates a new node containing this key-value, inserts it into the primary linked list and inserts a pointer to this node at the end of the aux list. The primary sends the key, the new value to be added and the old value associated with the key to all the fused backups. Each fused backup maintains a stack that contains the primary elements in the coded form. On receiving the insert update from X_i, if the key is not already present, the backup updates the code value of the fused node following the one that contains the top-most element of X_i Further to maintain order information, the backup inserts a pointer to the newly updated fused node, into the aux structure for X_i with the key received.

Figure 3(ii) shows the state of X_1 and F_1 after the insert of $(3, a_1^*)$. We assume that the keys are sorted in this linked list and hence the key-value pair $(3, a_1^*)$ is inserted at index 1 of the primary linked list and a pointer to a_1^* is inserted at the end of the aux list. At the backup, the value of the second node is updated to $a_1^* + b_3$ and a pointer to this node is inserted at index 1 of the aux linked list for X_1. The identical operation is performed at F_2, with the only difference being that the second fused node is updated to $a_1^* - b_3$. Observe that the aux list at the primary X_1 in figure 3(ii) specifies the exact order of elements maintained at the backup stack ($a_1 \rightarrow a_2 \rightarrow a_1^*$). Analogously, the aux list for X_1 at the fused backup points to the fused nodes that contain elements of X_1 in the correct order ($a_1 \rightarrow a_1^* \rightarrow a_2$).

Delete: Figure 5 shows the algorithms for the delete of a key at the primaries and the backups. X_i deletes the node associated with the key from the primary data structure and obtains its value which needs to be sent to the backups. Along with this value and the key k, the primary also sends the value of the element pointed to by the tail node of the aux list. This corresponds to the top-most element of X_i at the backup stack and is hence required for the shift operation that will be performed at the backup. After sending these values, the primary shifts the final node of the aux list to the position of

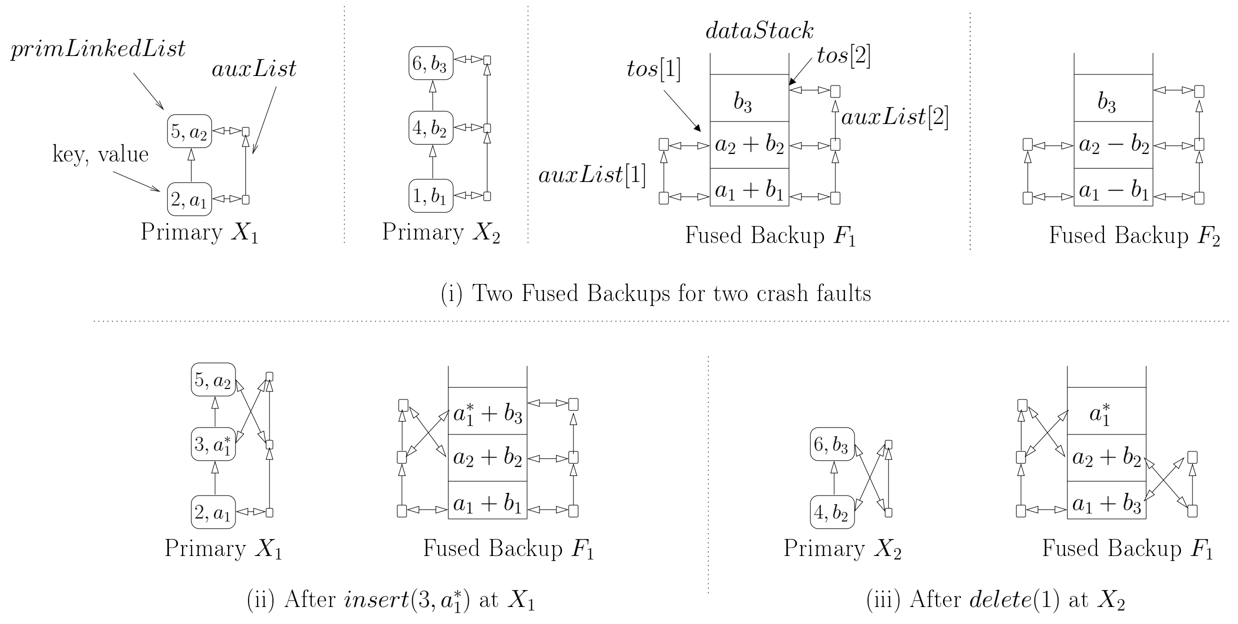


Fig. 3. Fused Backups for Linked Lists

```
INSERT at Primaries $X_i :: i = 1..n$
    Input: key $k$, data value $d$;
    if ($primLinkedList \cdot contains(k)$)
        /* key present, just update its value*/
        $old = primLinkedList \cdot get(k) \cdot value$
        $primLinkedList \cdot update(k, p)$;
        $send(k, d, old)$ to all fused backups;
    else
        /* key not present, create new node*/
        primNode $p$ = new primNode;
        $p \cdot value = d$;
        auxNode $a$ = new auxNode;
        $a \cdot primNode = p$;
        $p \cdot auxNode = a$;
        /* mimic backup stack */
        $auxList.insertAtEnd(a)$;
        $primLinkedList \cdot insert(k, p)$;
        $send(k, d, null)$ to all fused backups;
```

```
INSERT at Fused Backups $F_j :: j = 1..t$
    Input: key $k$, new value $d_i$, old value $old_i$;
    if ($auxLinkedList[i] \cdot contains(k)$)
        fusedNode $f = auxLinkedList[i] \cdot get(k)$;
        $f \cdot updateCode(old_i, d_i)$;
    else
        fusedNode $p = tos[i] + +$;
        if ($p == null$)
            $p$ = new fusedNode;
            $dataStack \cdot insert(p)$;
            $dataStackTos + +$;
        $p \cdot updateCode(0, d_i)$;
        $p \cdot refCount + +$;
        /* mimic primary linked list */
        auxNode $a$ = new auxNode;
        $a \cdot fusedNode = p$;
        $p \cdot auxNode[i] = a$;
        $auxLinkedList[i] \cdot insert(k, a)$;
```

Fig. 4. Fused Backups for Linked Lists: Inserts

the aux node pointing to the deleted element, to mimic the shift of the final element at the backup.

On receiving these values from the primary, the backup first obtains the fused node p pointed to by the aux node associated with k in the auxiliary structure of X_i at the backup. As the aux structure of X_i at the backup preserves the exact position information of the elements of X_i, p contains the element of X_i associated with k. The backup updates the value of p with the top-most element (sent by the primary as tos) to simulate the shift. The aux node pointers are updated to reflect this shift. Figure 3(iii) shows the state of X_1 and F_1 after the delete of b_1. The key things to note are the fact that at F_1, b_3 has been shifted from the end to the 0^{th} node, the aux list at X_2 reflects the correct order of its elements at the backup stack ($b_3 \rightarrow b_2$) and the aux structure at F_2 also reflects the correct order of elements at X_1 ($b_2 \rightarrow b_3$). Note that, the space and time overhead of maintaining the auxiliary list at the primary is negligible.

B. Fused Backups for Complex Data Structures

The design of fused backup for linked lists can easily be generalized for all types of data structures. At each primary along with the primary data structure, we maintain an auxiliary list that tracks the order of elements at the backup stack. At each backup, we maintain auxiliary structures for each primary, which is identical to the corresponding primary except for the fact that it has pointers to the fused nodes rather

```
DELETE at Primaries $X_i :: i = 1..n$
    Input: key $k$;
    $p = primLinkedList \cdot delete(k)$;
    $old = p \cdot value$;
    /* tail node of aux list points to top-most
        element of $X_i$ at backup stack */
    auxNode $auxTail = auxList \cdot getTail()$;
    $tos = auxTail \cdot primNode \cdot value$;
    $send(k, old, tos)$ to all fused backups;
    auxNode $a = p \cdot auxNode$;
    /* shift tail of aux list to replace $a$ */
    $(a \cdot prev) \cdot next = auxTail$;
    $auxTail \cdot next = a \cdot next$;
    $delete\ a$;
```

```
DELETE at Fused Backups $F_j :: j = 1..t$
    Input: key $k$, old value $old_i$, end value $tos_i$;
    /* update fused node containing $old_i$
        with primary element of $X_i$ at $tos[i]$*/
    auxNode $a = auxLinkedList[i] \cdot delete(k)$;
    fusedNode $p = a \cdot fusedNode$;
    $p \cdot updateCode(old_i, tos_i)$;
    $tos[i] \cdot updateCode(tos_i, 0)$;
    $tos[i] \cdot refCount - -$;
    /* update aux node pointing to $tos[i]$ */
    $tos[i] \cdot auxNode[i] \cdot fusedNode = p$;
    if ($tos[i].refCount == 0$)
        $dataStackTos - -$;
    $tos[i] - -$;
```

Fig. 5. Fused Backups for Linked Lists: Deletes

than primary elements. We explain this using the example of balanced binary search trees (BBST). Figure 6(i) shows two primary BBSTs and a fused backup. For simplicity, we explain the design using just one backup. The auxiliary structure at F_1 for X_1 is a BBST containing a root and two children, identical in structure to X_1. The algorithms for inserts and deletes at both primaries and backups remains identical to linked lists except for the fact that at the primary, we are inserting into a primary BBST and similarly at the backup we are inserting into an auxiliary BBST rather than an auxiliary linked list. Figure 6(ii) shows the state of X_1 and F_1 after the delete of a_3 followed by the insert of a_4. The aux list at X_1 specifies the order $(a_1 \rightarrow a_2 \rightarrow a_4)$, which is the order in which the elements of X_1 are maintained at F_1 in figure 6(ii). Similarly, the auxiliary BBST for X_1 at F_1 maintains the ordering of the elements at X_1. Since the root at X_1 is the element containing a_1, the root of the aux BBST at F_1 points to the fused node containing a_1. As we maintain auxiliary structures at the backup that are identical to the primary data structures, it is is not necessary that each container provide the semantics of $insert(key, value)$ and $delete(key)$. For example, we can also support the semantics $insert(position, value)$ and $delete(position, value)$ since the primary data structures and the auxiliary data structure being identical, support them.

So far we have focused only on the insert and delete operations to the data structure, since those are the operations that are adding and deleting data nodes. However, since we maintain the entire index structure at the backups, we support any operation to the primary data structure, as long as the corresponding operation to the backup does not involve decoding the values of the primary elements at the backup. We illustrate this with the example of the balance operation in the BBST shown in figure 6(iii). The balance at the primary just involves a change in the relative ordering of the elements. The update corresponding to this at the fused backup will change the relative ordering of the elements in the auxiliary BBST, identical to that at the primary. In conclusion, our design for fused backups can support all types of data structures with many complex operations. Based on this we have implemented fusible data structures and primaries for linked lists, vectors, queues, hash tables, tree maps etc. In the following section we describe the algorithms for the detection and correction of crash/Byzantine faults that are common to all types of primaries.

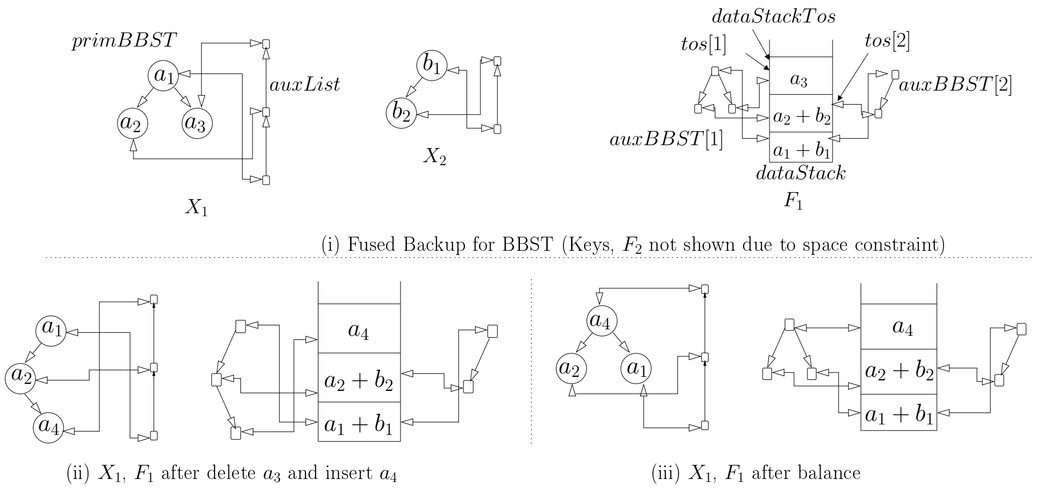
C. Fault Detection and Correction

To correct crash faults, we need to obtain all the available data structures, both primaries and backups. As seen in section III, the fused node at the same position at all the fused backups are the codewords for the primary elements belonging to these nodes. To obtain the missing primary elements belonging to this node, we decode the code words of these nodes along with the data values of the available primary elements belonging to this node. We apply the standard erasure decoding algorithm for decoding each set of values. In figure 3(i), to recover the state of the failed primaries, we obtain F_1 and F_2 and iterate through their nodes. The 0^{th} fused node of F_1 contains the value $a_1 + b_1$, while the 0^{th} node of F_2 contains the value $a_1 - b_1$. Using these, we can obtain the values of a_1 and b_1. The value of all the primary nodes can be obtained this way and their order can be obtained using the index structure at each backup.

To correct Byzantine faults, the only difference is that we decode the codes for errors rather than erasures. To detect Byzantine faults, we need to periodically encode the values of the primaries and compare it to the fused values at the backup. If these values do not match, this indicates a Byzantine error. In general, a code that can correct f erasures can detect f errors and correct $\lfloor f/2 \rfloor$ errors [2], [13], [19]. Hence, the fused backups that can correct f crash faults can also detect f Byzantine faults and correct $\lfloor f/2 \rfloor$ Byzantine faults. In the following section, we describe Reed Solomon codes as the fusion operator for f fault tolerance.

D. Reed Solomon Codes as Fusion Operator

In this section, we present the Reed Solomon (RS) erasure codes that can be used as a fusion operator to correct f faults



primBBST
auxList
a_1
a_2
a_3
X_1
b_1
b_2
X_2
dataStackTos
tos[1]
tos[2]
auxBBST[2]
auxBBST[1]
a_3
a_2+b_2
a_1+b_1
dataStack
F_1
(i) Fused Backup for BBST (Keys, F_2 not shown due to space constraint)
a_4
(ii) X_1, F_1 after delete a_3 and insert a_4
(iii) X_1, F_1 after balance

Fig. 6. Fused Backups for Balanced Binary Search Trees

among the primaries using f backups. Readers are referred to standard texts on coding theory [2], [13], [19] for a thorough treatment. Given n data words $\{d_1, d_2, \dots d_n\}$, RS erasure coding generates f checksum words $\{c_1, c_2, \dots c_f\}$ that can correct f erasures among the data and the checksum words. All operations are performed over a finite field with more than $n+f$ elements [19]. Hence, we can use RS codes to fuse the primary elements of the data structures and the backup stacks maintain these f codewords.

Fusion (Encoding): The algorithm comprises of generating an $n \times (n+f)$ information dispersal matrix B, that satisfies the following properties:

- The $n \times n$ matrix in the first n columns is an identity matrix.
- Any sub-matrix formed by the deletion of f columns of the matrix, is invertible.

Hence, B can be represented as the combination of an identity matrix I and another matrix S, i.e., $B = \begin{bmatrix} I & S \end{bmatrix}$. B is derived from a Vandermonde matrix with elementary matrix operations. Let D be the data vector and P the encoded vector obtained after multiplying D with B, i.e., $\begin{bmatrix} D \end{bmatrix} \times \begin{bmatrix} B \end{bmatrix} = \begin{bmatrix} D \end{bmatrix} \times \begin{bmatrix} I & S \end{bmatrix} = \begin{bmatrix} P \end{bmatrix} = \begin{bmatrix} D & C \end{bmatrix}$, where C is the set of check sums (the fused data) computed for the data set D.

Update: Whenever a data word d_i is updated to d_i', all the code words can be updated just using the difference $d_i' - d_i$ and c_j:

$$c_j' = c_j + b_{j,i}(d_i' - d_i)$$

where $b_{j,i}$ is $(j,i)^{th}$ element of the information dispersal matrix B. Since the new code word is computed without the value of the other code words, updates are very efficient in RS erasure coding. This update corresponds to the $updateCode$ routine used in figures 4 and 5.

Recovery (Decoding): In the case of erasures, we can recover the data words using the encoded vector P and the information dispersal matrix B. Data word erasures are reflected by deleting the corresponding columns from B and P to obtain B' and P' that adhere to the equation, $D \times B' = P'$. When exactly f data words fail, B' is a $n \times n$ matrix. As mentioned above, any sub-matrix generated by deleting f columns from B is an invertible matrix. Hence, matrix B' is guaranteed to be invertible. The data words can be generated as follows: $P' \times (B')^{-1} = D$.

Time Complexity Analysis for Crash Correction: We consider the cost of crash fault recovery for fused backups based on section III-C with RS codes as the fusion operator. To recover the state of f failed structures among the $n+f$ primaries and fused backups, we need to obtain the state of the remaining n data structures at a centralized stateless server. For simplicity, we assume faults only in the primary lists. To obtain the data values of the failed lists belonging to a node in the same position at all the fused lists, we decode the f code words of these nodes along with the data values of the $n-f$ available primary lists belonging to this node. We iterate through all the nodes to obtain the entire failed lists. The time complexity of recovery is proportional to the number of nodes in each fused list, multiplied by the cost of RS decoding. Given n data values, the cost of recovering f values , each of size s by RS decoding is $O(sf^2n)$ [20]. Since the number of nodes in the fused list is bound by the size of the primary list, m, the time complexity for recovery is $O(msf^2n)$ where each primary has $O(m)$ nodes of $O(s)$ size each.

IV. Theory of Fused Data Structures

In this section we prove theoretical properties on the fused backups such as size optimality, update optimality, update order independence and so on, all of which are important considerations when implementing a system using these backups. These properties ensure that the overhead in space and time caused due to these backups is minimal. The results in this section apply for all types of primaries and are independent of the fusion operator used. The only assumption we make is that the codes can be updated locally in constant time (like RS codes).

A. *Space Optimality*

Consider n primaries, each containing $O(m)$ nodes, each of size $O(s)$. In [9], to correct one crash fault, the backup for linked lists and list-based queues consumes $O(nms)$ space, which is as bad as replication. We show that the fused backups presented in this paper require only $O(ms)$ space. Further, to correct f faults, we show that the fused backups need only $O(msf)$ space, which we prove is optimal. Replication, on the other hand requires $O(mnsf)$ space, which is $O(n)$ times more than fusion. In figure 3, the number of fused nodes in F_1 or F_2 is always equal to the number of nodes in the largest primary. When we insert a node at the backup, we do so at the end of the stack and when we create a hole due to a delete, we shift the final element of that primary in the stack to fill the hole. This ensures that the number of nodes in the stack never exceeds the number of nodes in the largest primary.

Lemma 1: The data stack of each fused backup contains only m fused nodes.

Proof: This is a data stack where we insert primary elements at the top of the stack. Clearly, if there are no holes in the data stack, i.e a fused node that does not contain an element from a primary followed by a fused node that does contain an element from the same primary, then the invariant holds. When the stack is empty, there are no holes. In the case of inserts to X_i, we always insert at the fused node on top of the last fused node containing an element from X_i. Hence, no hole is created. For deletes, when a hole is created, we shift the final element of the primary, pointed to by $tos[i]$ plug this hole, thereby maintaining the invariant. ■

In section III-D, we saw that using Reed Solomon codes, we can correct f crash faults among the primaries using just f fused backups. This is clearly an optimal number for the number of backups required to correct f crash faults. From lemma 1 we know that the fused backups contains just the maximum number of nodes across all primaries. Since the size of each primary element is $O(s)$, the backups space required by our fused backups with RS codes as the fusion operator is $O(msf)$. In the following theorem, we show that this is optimal when the data across the primaries is entirely uncorrelated.

Theorem 1 (Space Optimality): The fused backups generated by our design using RS codes as the fusion operator are of optimal size.

Proof: f crash faults among the primaries will result in the failure of at least f data nodes, each of size $O(s)$. To correct f crash faults among them, we need to maintain at least f backup nodes each of size $O(s)$. Since the data structures each contain $O(m)$ nodes, to correct f crash faults among them, we need to maintain f backups containing each containing $O(ms)$ space. Hence the minimum space required is $O(msf)$. The fused backups using RS codes consumes exactly that amount of backup space and is hence optimal. ■

B. *Efficient Updates*

We define update optimality as follows: the time complexity of updates to the backup for all operations takes the same time as that of the corresponding update to the primary. In [9], in order to update the backup for linked lists, we need to iterate through all the fused nodes. Since the number of fused nodes in the backup is $O(nm)$, the time complexity of updates is $O(nm)$, while the time complexity for the update at the primary is only $O(m)$. Hence, it is not update optimal. We show that the fused backups presented in this paper are update optimal for all types of primaries, thereby causing minimal overhead during normal operation. An update to the backup consists of two operations: updating the data stack and updating the auxiliary structure. The first takes constant time while the second, which involves inserting or deleting into the auxiliary structure, takes only as much time as that at the primary since the auxiliary structure is constructed to be identical in structure to the primary data structure.

Theorem 2 (Update Optimality): The time complexity of the updates to a fused backup is of the same order as that at the primary.

Proof: Let us assume primary X_i sends an insert or delete of an element with key k. Operations at the backup consist of update to the data stack and the update to the auxiliary structure corresponding to the primary that sent the update, say X_i. In the case of inserts, we obtain the node following the top most element of X_i in the data stack and update it in constant time. The update to the auxiliary structure consists of an insert of an element with key k, which is the identical operation at the primary.

Similarly, for deletes, we first remove the node with key k from the auxiliary data structure, an operation that was executed on the data structure of the same type at the primary. Hence, it takes as much time as that at the primary. Once we obtain the auxiliary node that was deleted, we obtain a pointer to the fused node that needs to be updated followed by a constant time shift of the final element to this position. So, for both inserts and deletes the fused backups are update optimal. This extends easily for more complex information, because the basic idea is the same: any complex operation performed on the index structure of the primary will also be performed on the index structure maintained at the auxiliary structures at the backup. Updating the data nodes of the stack takes constant time. ■

Since the primaries are independent of each other, in many cases the updates to the backup can be to different fused nodes. Hence, multiple concurrent threads updating each fused backup can achieve considerable speed-up as long as the overhead of locking is minimal. As described in section III, the updates to the fused backup corresponding to a primary are mostly independent of the elements belonging to the other primaries and they do not affect their order at the backups. In the following theorem we show that multiple threads belonging to different primaries can update the fused backup concurrently with minimal locking of nodes.

Theorem 3 (Concurrent Updates): There exists an algorithm for multiple threads belonging to different primaries to update a fused backup concurrently with only $O(1)$ locking of nodes.

Proof: We modify the algorithms in figures 4 and 5 to enable concurrent updates. We assume the presence of fine grained locks that can lock just the fused nodes and if required a fused node along with the $dataStackTos$. Since updates from the same primary are never applied concurrently, we don't need to lock the index structure.

Inserts: If the insert has to create a new fused node, then the updating thread has to lock $dataStackTos$ and the fused node pointed to by this pointer using a single lock, insert and update a new fused node, increment $dataStackTos$ and then release this combined lock. If the insert from X_i does not have to create a new node it only has to lock the node pointed to by $tos[i]$, update the node's code value and release the lock. When the primaries are of different sizes, then the insert to the backups never occurs to the same fused node and hence are fully concurrent.

Deletes: The updating thread has to obtain the node containing the element to be deleted, lock it, update its value with tos and release it. Then it has to lock the node pointed to by $tos[i]$, update it with zero and release the lock. Similar to the case of inserts, when the delete causes a node of the stack to be deleted, the thread needs to lock the $dataStackTos$ as well as the node pointed to by this pointer in one lock, delete the node, update the pointer and then release the combined lock. ■

C. Order Independence

The state of the fused backups in [9] are dependent on the ordered in which the updates are received. If we simply extend these algorithms for f-fault tolerance using erasure codes, then all the backups have to receive the updates in the same order failing which the fused nodes at the same position across the backups can contain different primary elements. In this case, recovery is not possible. For example in figure 3, if the first node of F_1 contains $a_1 + b_1$, while the first node of F_2 contains $a_2 - b_1$, then we cannot recover the primary elements when X_1 and X_2 fail. This implies the need for synchrony, that will cause considerable overhead during normal operation. We show that the fused backups in this paper are order independent. As a simple example consider the two updates shown in figures 3. The updates to the auxiliary list commute since they are to different lists. As far as updates to the stack are concerned, the update from X_1 depends only on the last fused node containing an element from X_1 and hence is independent of the update from X_2 which does not change the order of elements of X_1 at the fused backup. Similarly the update from X_2 is to the first and third nodes of the stack immaterial of whether a_1^* has been inserted or not. Hence, they also commute.

Theorem 4 (Order Independence): The state of the fused backups after a set of updates is independent of the order in which the updates are received, as long as updates from the same primary are received in FIFO order.

Proof: Clearly, updates to the auxiliary structure commute. In the case of inserts, the only fused node that is updated is the one following the last fused node containing an element from the corresponding primary. Since this is independent of other primaries, this is order independent. Similarly, the delete and shift operation do not depend on other primaries. The update of the values are independent of other primaries and they affect the same fused node independent of the order. Hence the updates commute. Other operations (other than insert and delete) affect only the auxiliary structure and hence they too commute. ■

D. Fault Tolerance with Limited Backup Servers

So far we have assumed that the primary and backup structures reside on independent servers for the fusion-based solution (the same is true for [9]). In many practical scenarios, the number of servers available maybe less than the number of fused backups. In these cases, some of the backups have to be distributed among the servers hosting the primaries. Given a set of n data structures, each residing on a distinct server, we prove that $\lceil n/(n+a-f) \rceil \cdot f$ backups are necessary and sufficient to correct f crash faults among the host servers, when there are only a additional servers available to host the backup structures. Further, we present an algorithm for generating the optimal number of backup structures. Based on the design in section III-D, we assume that we can correct f crash faults among the primaries using just f fused backups.

To simplify our discussion, we start with the assumption that *no* additional servers are available for hosting the backups. As some of the servers host more than one backup structure, f faults among the servers, results in more than f faults among the data structures. Hence, a direct fusion-based solution cannot be applied to this problem. Given a set of five primaries, $\{X_1 \ldots X_5\}$, each residing on a distinct server labelled, $\{H_1 \ldots H_5\}$, consider the problem of correcting three crash faults among the servers ($n = 5$, $f = 3$). Let us just generate three backups F_1, F_2, F_3, and distribute them among the hosts H_1, H_2, H_3 respectively. Crash faults among these three servers will result in the crash of six data structures, whereas these set of backups can only correct three crash faults. We solve this problem by partitioning the set of primaries and generating backups for each individual block.

In this example, we can partition the primaries into three blocks $[X_1, X_2]$, $[X_3, X_4]$ and $[X_5]$ and generate three fused backups for each block of primaries. Henceforth, we denote the backup obtained by fusing the primaries $X_{i_1}, X_{i_2}, \ldots$, by $F_j(i_1, i_2, \ldots)$. For e.g., the backups for $[X_1, X_2]$ are denoted as $F_1(1,2) \ldots F_3(1,2)$. Consider the following distribution of backups among hosts:

$$H_1 = [X_1, F_1(3,4), F_1(5)], H_2 = [X_2, F_2(3,4), F_2(5)]$$

$$H_3 = [X_3, F_1(1,2)], F_3(5)], H_4 = [X_4, F_2(1,2)]$$

$$H_5 = [X_5, F_3(1,2), F_3(3,4)]$$

The backups for any block of primaries, do not reside on any of the servers hosting the primaries in that block. Three server faults will result in at most three faults among the primaries belonging to any single block and its backups. Since the fused backups of any block correct three faults among the data structures in a block, this partitioning scheme can correct three server faults. Here, each block of primaries requires at least three distinct servers (other than those hosting them) to host their backups. Hence, for $n = 5$, the size of any block in this partition cannot exceed $n - f = 2$. Based on this idea, we present the following algorithm, to correct f faults among the host servers.

(*Partitioning Algorithm*): Partition the set of primaries X as evenly possible into $\lceil n/(n-f) \rceil$ blocks, generate the f fused backups for each such block and place them on distinct servers not hosting the primaries in that block.

The number of blocks generated by the partitioning algorithm is $\lceil n/(n-f) \rceil$ and hence, the number of backup structures required is $\lceil n/(n-f) \rceil \cdot f$. Replication, on the other hand requires $n \cdot f$ backup structures which is always greater than or equal to $\lceil n/(n-f) \rceil \cdot f$. We show that $\lceil n/(n-f) \rceil \cdot f$ is a tight bound for the number of backup structures required to correct f faults among the servers. For the example where $n = 5$, $f = 3$, the partitioning algorithm requires nine backups. Consider a solution with eight backups. In any distribution of the backups among the servers, the three servers with the maximum number of data structures will host nine data structures in total. For example, if the backups are distributed as evenly as possible, the three servers hosting the maximum number of backups will each host two backups and a primary. Failure of these servers will result in the failure of nine data structures. Using just eight backups, we cannot correct nine faults among the data structures. In the following theorem, we prove this result for general n and f along with the assumption that there are a (rather than zero) additional servers available to host the backup structures.

Theorem 5: Given a set of n data structures, each residing on a distinct server, to correct f faults among the servers, it is necessary and sufficient to add $\lceil n/(n+a-f) \rceil \cdot f$ backup structures.

Proof:

(Sufficiency):

We prove the correctness of the partitioning algorithm. Since the maximum number of primaries in any block of the partitioning algorithm is $n+a-f$, there are at least f distinct servers (not hosting the primaries in the block) available to host the f fused backups of any block of primaries. So, the fused backups can be distributed among the host servers such that f server faults only lead to f faults among the backups and primaries corresponding to each block. Hence the fused backups generated by the partitioning algorithm can correct f server faults.

(Necessity):

Suppose there is a scheme with t backups such that $t < \lceil n/(n+a-f) \rceil \cdot f$. In any distribution of the backups among the servers, choose f servers with the largest number of backups. We claim that the total number of backups in these f servers is strictly greater than $t - f$. Failure of these servers, will result in more than $t - f + f$ faults (adding faults of f primary structures). This would be impossible to correct with t backups. We know that,

$t < \lceil n/(n+a-f) \rceil \cdot f$
$\Rightarrow t < \lceil 1 + f/(n+a-f) \rceil \cdot f$
$\Rightarrow (t-f) < \lceil f/(n+a-f) \rceil \cdot f$
$\Rightarrow (t-f)/f < \lceil f/(n+a-f) \rceil$

If the f servers with the largest number of backups have less than or equal to $t - f$ backups in all, then the server with the smallest number of backups among them will have less than the average number of backups which is $(t-f)/f$.

Since the remaining $n + a - f$ servers have more than or equal to f backups, the server with the largest number of backups among them will have as many or greater than the average number of backups which is $\lceil f/(n+a-f) \rceil$.

Since, $(t-f)/f < \lceil f/(n+a-f) \rceil$, we get a contradiction that the smallest among the f servers hosting the largest number of backups , hosts less number of backups than the largest among the remaining $n - f$ servers. ■

V. Practical Example: Amazon's Dynamo

In this section, we present a practical application of our technique based on a real world implementation of a distributed system. Amazon's Dynamo [6] is a distributed data store that needs to provide both durability and very low response times (availability) for writes to the end user. They achieve this using a replication-based solution which is simple to maintain but expensive in terms of space. We propose an alternate design using a combination of both fused backups and replicas, which consumes far less space, while guaranteeing nearly the same levels of durability and availability.

A. Existing Dynamo Design

We present a simplified version of Dynamo with a focus on the replication strategy. Dynamo consists of clusters of primary hosts each containing a data store like a hash table that stores key-value pairs. The key space is partitioned across these hosts to ensure sufficient load-balancing. For both fault tolerance and availability, f replicas of each primary are maintained in distinct backup hosts. These $f+1$ identical copies can correct f crash faults among the primaries. The system also defines two parameters r and w which denote the minimum number of hosts that must participate in each read request and write request respectively. These values are each chosen to be less than f. The authors in [6] mention that the most common values of (n, w, r) observed among their clients are $(3, 2, 2)$. In figure 7, we illustrate a simple set up of dynamo for $n = 4$ primaries, with $f = 3$ replicas maintained for each one of them.

To read and write from the data store, the client can send its request to any one of the $f+1$ replicas that cater to the key of the request, and designate it the *coordinator*. The coordinator reads/writes the value corresponding to the key locally and sends the request to the remaining f copies. On receiving $r-1$ or $w-1$ responses from the replicas for read and write requests respectively, the coordinator responds to the client with the data value (for reads) or just an acknowledgment (for writes). Since $w < f$, clearly some of the replicas may not be up to date when the coordinator responds to the client. This necessitates some form of data versioning, and the coordinator or the client has to reconcile the different data versions on every read. This is considered an acceptable cost since Dynamo is mainly concerned with optimizing writes to the store. In this setup, when one or more hosts crash, the remaining copies that cater to the same key space can take over all requests addressed to the failed hosts. Once the crashed host comes back, the replica that was acting as proxy just transfers back the keys that were meant for the node. Since there can be at most three crash faults in the system, there is at least one node replica for each primary remaining for recovery.

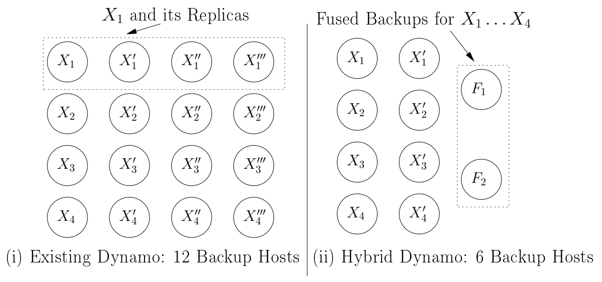

X_1 and its Replicas
Fused Backups for $X_1 \ldots X_4$
X_1 X_1' X_1'' X_1'''
X_2 X_2' X_2'' X_2'''
X_3 X_3' X_3'' X_3'''
X_4 X_4' X_4'' X_4'''
X_1 X_1' X_2 X_2' X_3 X_3' X_4 X_4' F_1 F_2
(i) Existing Dynamo: 12 Backup Hosts
(ii) Hybrid Dynamo: 6 Backup Hosts

Fig. 7. Design Strategies for Dynamo

B. Hybrid Dynamo Design

We propose a hybrid design for Dynamo that uses a combination of fusion and replication. We focus on the typical case of $(f, w, r) = (3, 2, 2)$. Instead of maintaining three replicas for each primary ($f = 3$), we maintain just a single replica for each primary and two fused backups for the entire set of primaries as shown in figure 7(ii). The fused backups, being optimal in size, achieve savings in space while the replicas allows the necessary availability for reads. The fused backups along with the replica can correct three crash faults among the primaries. The basic protocol for reads and writes remains the same except for the fact that the fused copies cannot directly respond to the client requests. In the case of writes, the fused copies require the old value associated with the key (section III). However, on receiving a write request, the coordinator can send the request to these fused backups which can respond to the request after updating the table. For the typical case of $w = 2$, as long as the coordinator, say X_i obtains a response from one among the three backups (a replica and two fused backups) the write can succeed. This is similar to the existing design and hence performance for writes is not affected by much. On the other hand, performance for reads does drop since the fused copies that contain data in the coded form cannot return the data value corresponding to a key in an efficient manner. Hence, the two replicas need to answer all requests to maintain availability. Since Dynamo is optimized mainly for writes, this may not be a cause for concern. To alleviate the load on the fused backups, we can partition the set of primaries into smaller blocks. For the set up shown in figure 7, we can maintain four fused backups where F_1, F_2 are the fused copies for X_1 and X_2, while F_3 and F_4 are the fused copies of X_3 and X_4. This will reduce the load on the backups while ensuring that the number of hosts as compared to the solution in figure 7(ii) increases only by two.

Similar to the existing design, when hosts crash, if there are surviving replicas that cater to the same keys, then they can take over operation. However, since we maintain only one host replica per primary, it is possible that none of the replicas remain. In this case, the fused backup can *mutate* into one or more of the failed primaries. It can receive requests corresponding to the failed primaries, update its local hash table and maintain data in its normal form (without fusing them). Concurrently, to recover the failed primaries, it can obtain the data values from the remaining hosts and decode the values. Hence, even though transiently the fault tolerance of the system is reduced, there is not much reduction in operational performance. Dynamo has been designed to scale to 100 hosts. So in a typical cluster with $n = 100$, $f = 3$ the original approach requires, $n * f = 300$ backup hosts. Consider a hybrid solution that maintains a replica for each host and maintains two fused backups for every 10 hosts. This approach requires only $100 + 20 = 120$ backup hosts. This argument can be extended to all Dynamo clusters deployed around the world. The savings achieved in space, power and resources can be crucial for such a real-world system.

VI. Implementation and Results

In this section, we describe our fusion-based data structure library [1] that includes all data structures provided by the Java Collection Framework. Further we have evaluated our performance against replication and the older version of fusion [9]. The current version of fusion outperforms the older version on all three counts: Backups space, update time at the backups and time taken for recovery. In terms of comparison with replication, we save $O(n)$ times space as confirmed by the theoretical results while not causing too much update overhead. Recovery is much cheaper in replication.

Fault-Tolerant Data Structure Library : We implemented fused backups and primary wrappers for the data structures in the Java 6 Collection framework that are broadly divided into list-based, map-based, set-based and queue-based data structures. The fusion operator is RS codes based on the C++ library provided by James S. Plank [21]. We evaluated the performance of a representative data structure in each of these categories: linked lists for list-based, tree maps for map-based, hash sets for set-based and queues for queue-based data structures.

Evaluation: To evaluate performance, we implemented a distributed system of hosts, each running either a primary or a backup data structure and compared the performance of our solution ('New Fusion') with the one presented in [9] ('Old Fusion') and replication. The algorithms were implemented in

Java 6 with TCP sockets for communication and the experiments were executed on a single Intel quad-core PC with 2.66 GHz clock frequency and 12 GB RAM. The three parameters that were varied across the experiments were the number of primaries n, number of faults f and the total number of operations performed per primary, ops. The operations were biased towards inserts (80 %) and the tests were averaged over four runs. We describe the results for the three main tests that we performed: backup space, update time at the backup and recovery time. The graphs for these tests are shown in figures 8 and 9.

To measure backup space, we assume that the size of data far exceeds the overhead of the index structure and hence, we just plot the total number of backup nodes consumed by each solution. We fix $f = 3$, $ops = 500$ and vary n from 1 to 10. The new version of fusion for linked lists performs much better than both replication (almost n times) and the solution in [9] (almost $n/2$ times) because the number of nodes per backup never exceeds the maximum among the primaries. We use the same experiment to calculate the recovery time taken by the three approaches measured as the time taken to decode *after* obtaining the necessary data structures at a recovery agent. The new version of fusion performs much better than old fusion (almost $n/2$ times) for a similar reason: recovery in fusion involves iteration through all the nodes of each fused backup. The newer version contains considerably less number of nodes and hence performs better. The time taken for recovery by replication is negligible as compared to both other solutions. This is to be expected since recovery in replication requires just copying the failed data structures after obtaining them. Finally, to measure the update time at the backups, since we were primarily concerned with the overhead of coding, we fixed $n = 1$, $f = 1$ and varied ops from 500 to 5000. The new version of fusion has slightly more update overhead as compared to replication (around 1.25 times slower) while it performs much better than the older version (almost 3 times faster) since the latter solution may have to iterate through all the fused nodes at the backup for updates. The practical results hence confirm the theoretical bounds: the solution presented in this paper saves considerably on space, while causing minimal overhead during normal operation.

VII. Related Work

In our work in [15], we present a coding-theoretic solution to fault tolerance in finite state machines. This approach is extended for infinite state machines and optimized for Byzantine fault tolerance in [8]. Extensive work has been done on consensus in distributed systems. The FLP result [7] states that it is impossible to achieve consensus among a given set of machines in an asynchronous system with even one crash fault among the machines. We assume the presence of a failure detector for crash faults in our system. Similarly, there has been a considerable body of research for solving consensus among the servers in synchronous systems. Given a system of n machines in which up to f machines may undergo Byzantine faults, consensus cannot be achieved unless $n > 3f$ [17]. In the case of crash faults, at least $f + 1$ synchronous rounds are required to achieve consensus [11]. These results do not apply to our model because we assume a trusted recovery agent.

VIII. Conclusion

Given n primary data structures, we present a fusion-based technique that guarantees $O(n)$ savings in space as compared to replication and prove theoretically that they incur minimal overhead during normal operation. We provide a generic design of fused backups and their implementation for all the data structures in the Java 6 Collection framework that includes vectors, stacks, maps, trees and most other commonly used data structures. Our evaluation confirms that fusion is extremely space efficient while recovery is cheaper in replication. In a system with infrequent faults fusion is a better choice than replication. We compare the main features of our work with a previous work on this topic [9] and replication, in Table I. Many real world systems such Amazon's Dynamo or Google's Map-Reduce framework use replication extensively for fault tolerance. Using concepts presented in this paper, we can consider an alternate design using a combination of replication and fusion-based techniques. We illustrate this in section V by presenting a simple design alternative for Amazon's data store, Dynamo. In a typical Dynamo cluster of 100 hosts our combined approach requires only 120 backup hosts as compared to the existing set up of 300 backup hosts without compromising on other important QoS parameters such as response times. Thus fusion achieves significant savings in space, power and resources.

IX. Acknowledgment

We are thankful to Vinit Ogale for discussions on the topic. This research was supported in part by the NSF Grants CNS-0718990, CNS-0509024, Texas Education Board Grant 781, SRC Grant 2006-TJ-1426, and Cullen Trust for Higher Education Endowed Professorship.

References

[1] Bharath Balasubramanian and Vijay K. Garg. Fused data structure library (implemented in java 1.6). In *Parallel and Distributed Systems Laboratory, http://maple.ece.utexas.edu*, 2010.

[2] E. R. Berlekamp. *Algebraic Coding Theory*. McGraw-Hill, New York, 1968.

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

[4] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A digital fountain approach to reliable distribution of bulk data. In *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 56–67, New York, NY, USA, 1998. ACM.

[5] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. Raid: high-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, 1994.

[6] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

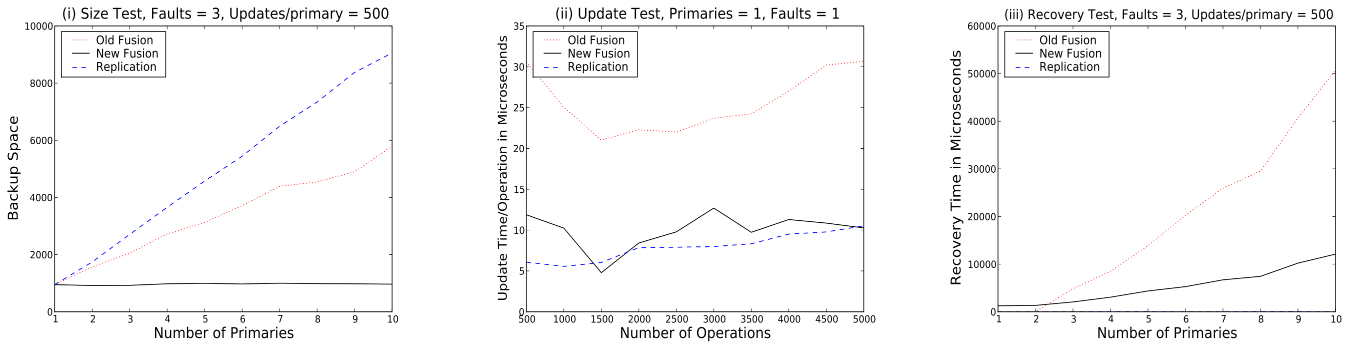


Fig. 8. Linked List Tests

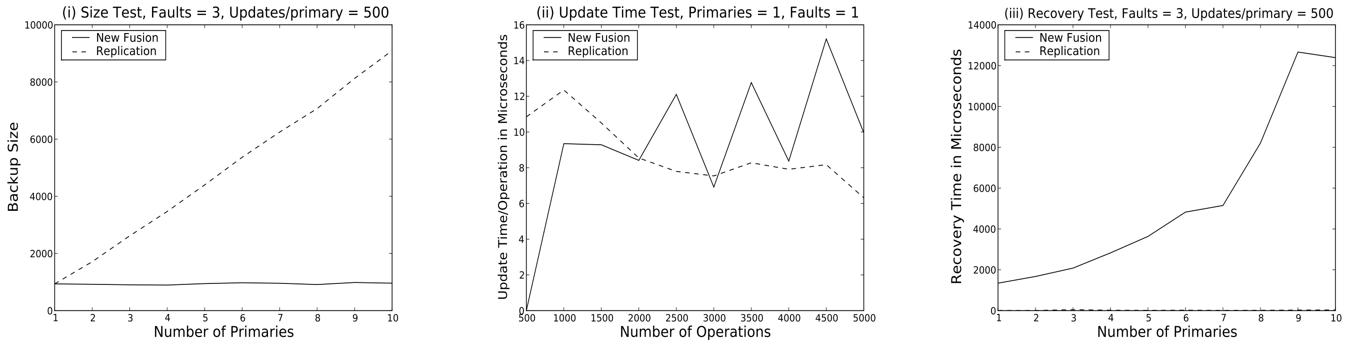


Fig. 9. Tree Map Tests

[7] M. J. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), April 1985.

[8] Vijay K. Garg. Implementing fault-tolerant services using state machines: Beyond replication. In *DISC*, pages 450–464, 2010.

[9] Vijay K. Garg and Vinit Ogale. Fusible data structures for fault tolerance. In *ICDCS 2007: Proceedings of the 27th International Conference on Distributed Computing Systems*, June 2007.

[10] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer networks*, 2:95–114, 1978.

[11] Leslie Lamport and Michael Fischer. Byzantine generals and transaction commit protocols. Technical report, 1982.

[12] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

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

[14] Michael G. Luby, Michael Mitzenmacher, M. Amin Shokrollahi, Daniel A. Spielman, and Volker Stemann. Practical loss-resilient codes. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 150–159, New York, NY, USA, 1997. ACM Press.

[15] Vinit Ogale, Bharath Balasubramanian, and Vijay K. Garg. A fusion-based approach for tolerating faults in finite state machines. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

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

[17] M. Pease and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27:228–234, 1980.

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

[19] Wesley W. Peterson and E. J. Weldon. *Error-Correcting Codes - Revised, 2nd Edition*. The MIT Press, 2 edition, March 1972.

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

[21] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2. Technical Report CS-08-627, University of Tennessee, August 2008.

[22] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM*, 36(2):335–348, 1989.

[23] I. S. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[24] Fred B. Schneider. Byzantine generals in action: implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, 1984.

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

[26] Claude Elwood Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423,623–656, 1948.

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