

An example is shown in fig.1 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.

4.2 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 a node 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. The result 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 0th 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.

4.3 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 than primary elements. 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.

As we maintain auxiliary structures at the backup that are identical to the primary data structures, it 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.

4.4 Fault Detection and Correction

To correct crash faults, we need to obtain all the available data structures, both primaries and backups. The fused nodes at the same position at all the fused backups are the code words 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. To recover the state of the failed primaries, we obtain F_1 and F_2 and iterate through their nodes. The 0th fused node of F_1 contains the value $a_1 + b_1$, while the 0th 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 $f/2$ errors. Hence, the fused backups that can correct f crash faults can also detect f Byzantine faults and correct $f/2$ Byzantine faults.

4.5 File System in Grid

File Systems for Grids Network file systems can substantially simplify using heterogeneous distributed storage resources in grid environments. They not only provide data sharing among multiple sites, but also allow the user to view multiple, distributed local file systems as a unified single file system. Below, we will discuss the major requirements for file systems on grid environments, including *security* and *scalability*.

NFS or NFS based file systems combined with security mechanisms support secure data sharing in wide area networks. However, these systems basically consist of only a single- or a fixed array of file server node(s), which often become a performance bottlenecks for large datasets; therefore, NFS-based file systems are typically inappropriate for grid environments in terms of scalability. Striping parallel file systems, which exploit the efficiency of using multiple storage nodes simultaneously, are mainly used in HPC cluster environments to attain fast I/O performance. All

the files are divided into fixed-size chunks, and each chunk can be placed on any storage node.

However, the performance of such file systems can often be limited by the available network bandwidth, which is abundant for local clusters but often would be poor for wide-area grid environments. Furthermore, most of these file systems do not support sufficient security mechanisms that are required to operate over multiple administrative domains.

Grid Data farm is file system architecture for peta-scale data-intensive computing on grid environments. It not only provides data sharing on grid environments, but also supports file-location-aware job scheduling—Jobs are automatically scheduled to execute on nodes where their required files or file segments are already available in their local storage. Such scheduling can be often an efficient heuristics because it can exploit local I/O performance, rather than transferring large data between compute and storage nodes. However, users need to manually manage data to improve I/O performance when heterogeneous resources are used. Moreover, inefficient file accesses and compute scheduling situations could easily arise, such as a large number of jobs are being simultaneously accessing the same file, causing substantial CPU and I/O contentions and thereby degrading the overall system throughput. Our intelligent data replication technique tackles these performance problems.

4.6 Algorithm Description

4.6.1 Insert Fused Backups

This algorithm is for the insert of a key-value pair at the primaries and the backups. When the client sends an insert to a primary X_i , if the key is not already present, X_i creates a new node containing this key value, inserts it into the primary linked list (denoted primaryLinkedList) and inserts a pointer to this node at the end of the aux list (auxList). 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 (data Stack) that contains the primary elements in the coded form. On receiving the insert from X_i , if the key is not already present, the backup updates the code value of the fused node following the one contains the top-most element of X_i (pointed to by $tos[i]$). To maintain order information, the backup inserts a pointer to the newly updated fused node, into the index structure (indexList[i]) for X_i with the key received. A reference count (refCount) tracking the number of elements in the fused node is maintained to enable efficient deletes.

Algorithm

- Step 1: initialize the linked list and Stack
- Step 2: Insert the backup into linked list
- Step 3: If replicas contains, insert replica data into stack
- Step 4: Get top of the stack data
- Step 5: Stored into linked list element

4.6.2 Delete Fused Backups

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

Algorithm:

- Step 1: Gather Top of the Stack
- Step 2: Move TOS into linked list
- Step 3: Store Linked list element
- Step 4: Clear Stack Elements
- Step 5: Set Stack is empty, Null is TOS

5. Experimental Results

Our proposed work fusion-based data structure library that includes all data structures provided by the Visual Studio Framework. Further we have evaluated our performance against replication and the older version of fusion. 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.

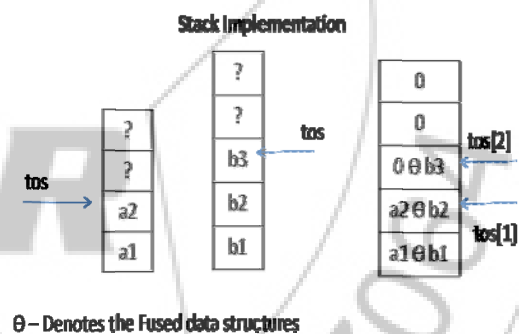


Figure 2: Array Based Stacks

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.

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.

Push Operation

```
function  $x_i$ .push(newItem)
 $x_i$ .array[ $x_i$ .tos] := newItem;
```

```

xi.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
    
```

Pop Operation

```

function xi:pop()
x.tos[i]--;
y.pop(i, xi.array[xi.tos]);
return xi.array[xi.tos]
function y:pop(i; oldItem)
y.tos[i]--;
y.array[y.tos[i]] := y.array[y.tos[i]] - oldItem;
end function
    
```

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, \dots, n_k which is less than N . Therefore, the space constraint is satisfied.

Recover Operation

```

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 - xp.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.

5.1 Performance Comparison With The Existing System

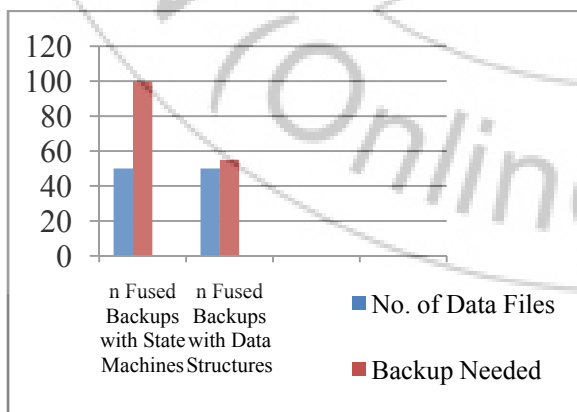


Figure 3 : Performance Comparisons with the Existing System

structures required by replication. For Byzantine faults, fusion requires $nf + f$ backups as compared to the $2nf$ backups required by replication. For crash faults, the total space occupied by the fused backups in msf as compared to $nmsf$ for replication (nf backups of size ms each). For Byzantine faults, since we maintain f copies of each primary along with f fused backups, the space complexity for fusion is $nfms + msf$ as compared to $2nmsf$ for replication.

5.2 Performance of Fused Backups

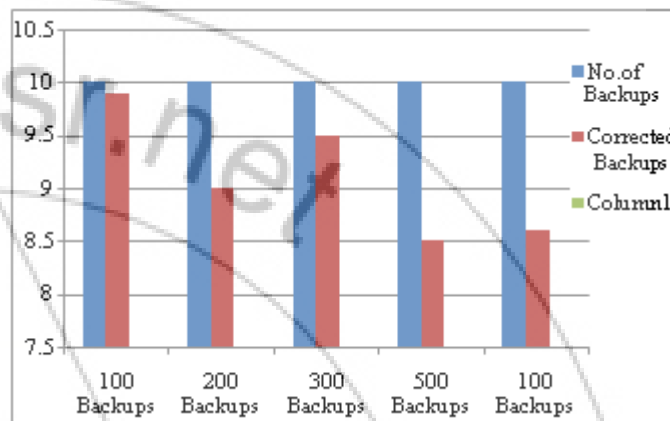


Figure 4: Performances of Fused Backups

This refers to the number of messages that need to be exchanged once a fault has been detected. When t crash faults are detected, in fusion, the client needs to acquire the state of all the remaining data structures. This requires $n-t$ messages of size $O(ms)$ each. In replication the client only needs to acquire the state of the failed copies requiring only t messages of size $O(ms)$ each. For Byzantine faults, in fusion, the state of all $n + nf + f$ data structures (primaries and backups) needs to be acquired. This requires $nf + f$ messages of size $O(ms)$ each. In replication, only the state of any $2t + 1$ copies of the faulty primary are needed, requiring just $2t + 1$ messages of size $O(ms)$ each.

5.3 Time Complexity of Fused Backups

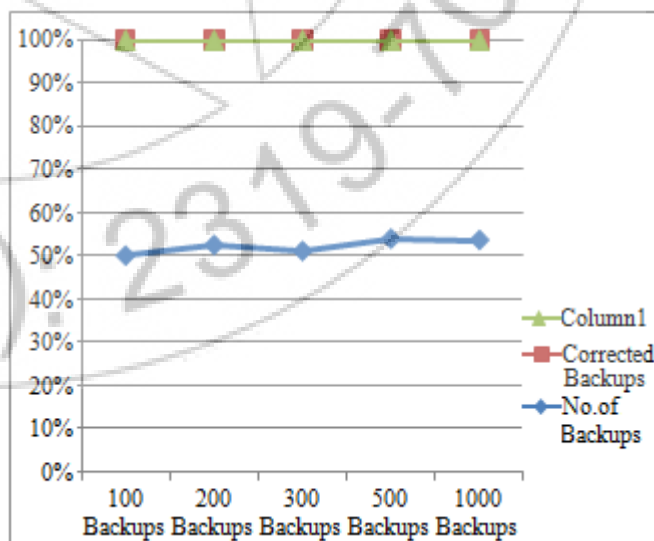


Figure 5: Time Complexity of Fused Backups

To correct f crash faults among n primaries, fusion requires f backup data structures as compared to the nf backup data

It defines the number of backups move from the different servers to the client also analysis the faulted and corrected

backup's performance. The chart defines different backups and corrected data transfer to the client machine.

6. Conclusion and Future Work

A fusion-based technique for fault tolerance was that savings in space as compared to replication with almost no overhead during normal operation. This System provide a generic design of fused backups and their implementation for all the data structures in the Visual Studio framework that includes vectors, stacks, maps, trees, and most other commonly used data structures. This System compare the main features of work with replication, both theoretically and experimentally. This work confirms that fusion is extremely space efficient while replication is efficient in terms of recovery, load on the backups and the size of the messages that need to be sent to the backups. In our future, we investigate the other data structure concepts like Queue, and Tree methods to implement the current system. Also increase the system performance when we transfer the bulk data from the server to client. Utilize the Main memory to recover the faulted data.

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] Bharath Balasubramanian and Vijay K. Garg. Fused data structures for handling multiple faults in distributed systems. In Proceedings of the 2011 31st International Conference on Distributed Computing Systems, ICDCS '11, pages 677–688, Washington, DC, USA, 2011. IEEE Computer Society.
- [3] Bharath Balasubramanian and Vijay K. Garg. Fused state machines for fault tolerance in distributed systems. In Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings, volume 7109 of Lecture Notes in Computer Science, pages 266–282. Springer, 2011.
- [4] Melliar-Smith P. M., Moser L. E, and Agrawala.v. Broadcast protocols for distributed systems. IEEE Trans. Parallel Distrib. Syst., 1(1):17–25, January 1990.
- [5] Ousterhout J.K., Agrawal. P., Erickson . D., Kozyrakis. C, Leverich .J, Mazie' res. D, Mitra. S, Narayanan A., Rosenblum M., Rumble. S.M., Stratmann. E., and Stutsman. R., "The Case for RAMClouds: Scalable High-Performance Storage Entirely in Dram," ACM SIGOPS Operating Systems Rev., vol. 43, pp. 92-105, 2009.
- [6] Patterson. D.A., Gibson. G., and Katz. R.H., "A Case for Redundant Arrays of Inexpensive Disks (RAID)," Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '88), pp. 109-116, 1988.
- [7] Peterson. W.W and Weldon E.J., Error-Correcting Codes - Revised, second ed. The MIT Press, Mar. 1972.
- [8] Plank. J.S., "A Tutorial on Reed-Solomon Coding for Fault- Tolerance in RAID-Like Systems," Software - Practice and Experience, vol. 27, no. 9, pp. 995-1012, Sept. 1997.
- [9] Plank J.S., Simmerman. S., and Schuman. C.D., "Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications - Version 1.2," Technical Report CS-08-627, Univ. of Tennessee, Aug. 2008.
- [10] Plank and Xu. L., "Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications," Proc. IEEE Fifth Int'l Symp. Network Computing and Applications, pp. 173-180, 2006.
- [11] Rabin M.O., "Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance," J. ACM, vol. 36, no. 2, pp. 335-348, 1989.
- [12] Reed I.S. and G. Solomon, "Polynomial Codes over Certain Finite Fields," J. Soc. for Industrial and Applied Math., vol. 8, no. 2, pp. 300-304, 1960.
- [13] Schneider. F.B., "Byzantine Generals in Action: Implementing Fail- Stop Processors," ACM Trans. Computer Systems, vol. 2, no. 2, pp. 145-154, 1984.
- [14] Schneider. F.B., "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," ACM Computing Surveys, vol. 22, no. 4, pp. 299-319, 1990.
- [15] Shannon. C.E., "A Mathematical Theory of Communication," Bell Systems Technical J., vol. 27, pp. 379-423 and 623-656, 1948.
- [16] Vijay K. Garg. Implementing fault-tolerant services using state machines: Beyond replication. In DISC, pages 450–464, 2010.