International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Impact Factor (2012): 3.358

Fault Tolerance in Parallel System Using Multiple Stacks

K. Meera[1], Prof. S. Abiramasundari[2]

[1]SASTRA University, Kumbakonam

[2]Assistant Professor, Department of Computer Science, SASTRA University, Kumbakonam

Abstract: ***The project describes a technique to tolerate faults in large data structures hosted on distributed servers, based on the concept of fused backups. The prevalent solution to this problem is replication. To tolerate the faults (dead/unresponsive data structures) among the whole distinct data structures, replication requires replicas of each data structure, resulting in number of servers and the number of fault for additional backups. This project present a solution, referred to as fusion that uses a combination of erasure codes and selective replication to tolerate f crash faults using just additional fused backups. This project shows that the solution achieves savings in space over replication. Further, this work present a solution to tolerate Byzantine faults (malicious data structures), that requires only backups as compared to the 2nf backups required by replication. We ensure that the overhead for normal operation in fusion is only as much as the overhead for replication. Though recovery is costly in fusion, in a system with infrequent faults, the savings in space outweighs the cost of recovery. This project explores the theory of fused backups and provides a library of such backups for all the data structures in the Visual Studio Collection Framework. The experimental evaluation confirms that fused backups are space-efficient as compared to replication (approximately n times), while they cause very little overhead for updates.***

Keywords: Data Structures, Fault Tolerance, Parallel Application, Stack**.**

1. Introduction

Parallel 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. These servers are prone to faults in which the data structures may crash, leading to a total loss in state (crash faults) or worse, they may behave in an adversarial manner, reflecting any arbitrary state, sending wrong conflicting messages to the client or other data structures (Byzantine faults).

Active replication is the prevalent solution to this problem. To tolerate f crash faults among n given data structures, replication maintains f + 1 replicas of each data structure, resulting in a total of nf backups. These replicas can also tolerate [f/2] Byzantine faults, since there is always a majority of correct copies available for each data structure [1].

2. Related Work

In [2], the theory of fused state machines uses a combination of coding theory and replication to ensure efficiency as well as savings in storage and messages during normal operations. Fused state machines may incur higher overhead during recovery from crash or Byzantine faults, but that may be acceptable if the probability of fault is low.

In [3], Fusible data structures satisfy three main properties: recovery, space constraint and efficient maintenance. The recovery property ensures that in case of a failure, the fused structure, along with the remaining original data structures, can be used to reconstruct the failed structure. The space constraint ensures that the number of nodes in the fused structures is strictly smaller than the number of nodes in the original structures. Finally, the efficient maintenance property ensures that when any of the original data structures is updated, the fused structure can be updated incrementally using local information about the update and does not need to be entirely recomputed.

In [4], Evaluation of fusion over standard benchmarks shows that efficient backups exist for many examples. To illustrate the practical use of fusion, we describe a fusion-based design of a distributed application in the Map Reduce framework. While the current replication-based solution may require 1.8 million map tasks, a fusion-based solution requires just 1.4 million map tasks with minimal overhead in terms of time as compared to replication. This can result in considerable savings in space and other computational resources such as power.

In [5], Dynamo, a highly available and scalable data store, used for storing state of a number of core services of Amazon.com's e-commerce platform. Dynamo has provided the desired levels of availability and performance and has been successful in handling server failures, data center failures and network partitions. Dynamo is incrementally scalable and allows service owners to scale up and down based on their current request load. Dynamo allows service owners to customize their storage system to meet their desired performance, durability and consistency SLAs by allowing them to tune the parameters N, R, and W.

In [6], RAIDS offer a cost effective option to meet the challenge of exponential growth of the processor and memory speed. This work believe the size reduction of personal computer disks is a key to the success of disk arrays, just as Gordon Bell argues that the size reduction of micro processors is a key to the success in multiprocessors. In both cases the smaller size simplifies the interconnection of

the many components as well as packaging and cabling While large arrays of mainframe processors (or SLEDS) are possible. it is certainly easier to construct an array from the same number of microprocessors (or PC drives) Just as Bell coined the term "multi" to distinguish a multiprocessor made from microprocessors, we use the term "RAID" to identify a disk array made from personal computer disks.

3. Proposed Work

The proposed system present a solution, referred to as fusion that uses to avoid replication. It shows that the solution achieves savings in space over replication. The fused backups are space-efficient as compared to replication (approximately n times), while they cause very little overhead for updates. In our proposed system, the data loss and time delay can be reduced when compared to the already existing services. Computer can carry pit calculation in just few seconds that would require months or perhaps even years when carried out by hand. Practically, the proposed system never makes a mistake of its own accord.

Advantages
- Avoid Replicas
- Less Backups
- Less Processing Time
- Low Space is enough
- Network Traffic is avoided
- Low cost comparing with existing system
- Router is used for boost up the network speed

4. Methodology Used

4.1 Parallel Computation

Parallel Computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). There are several different forms of parallel computing: bit-level, instruction level, data, and task parallelism. Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling. As power consumption (and consequently heat generation) by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors.

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism, with multi-core and processor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors, for accelerating specific tasks.

4.2 Insert Fused Backups

This algorithm 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

4.3 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

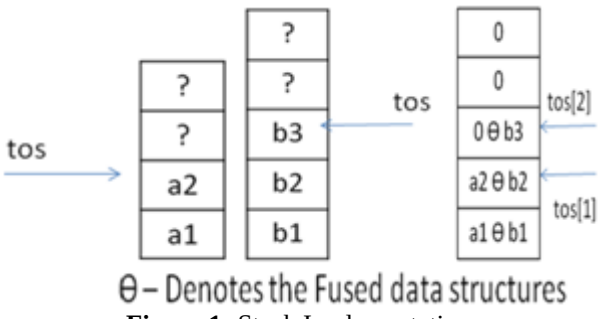

Figure 1: Stack Implementation

- In this system the stack will be fused when more than one replicated data files transfer to the client machine.
- 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.

Push Operation

function *xi:push*(*newItem*)
xi.array[*xi*.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

Recover Operation

function *y:recover*(*failedP rocess*)
/*Assuming that all source stacks have the same size*/
recoveredArray := new Array[*y*.array.size];
for *j* = 0 to *tos*[*failedP rocess*] ¡ 1
recItem := y[*j*];
foreach process *p != failedP rocess*
if (*j* < tos[p]) recItem := recItem - *xp*.array[*j*];
recoveredArray[*j*] := recItem;
return recoveredArray, tos[failedProcess]

Performance Comparison with the Existing System

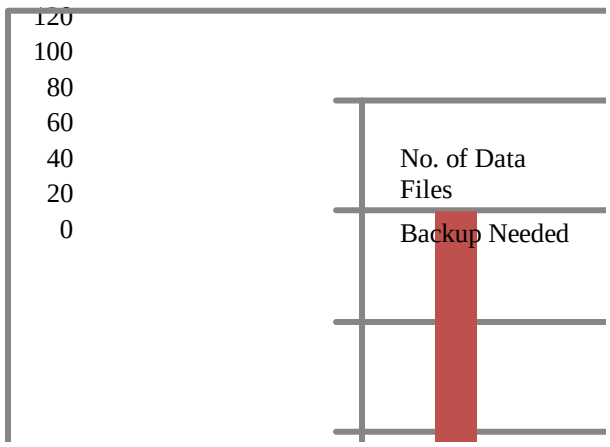

Figure 2: Performance Comparison with the Existing System

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

Performance of Fused Backups

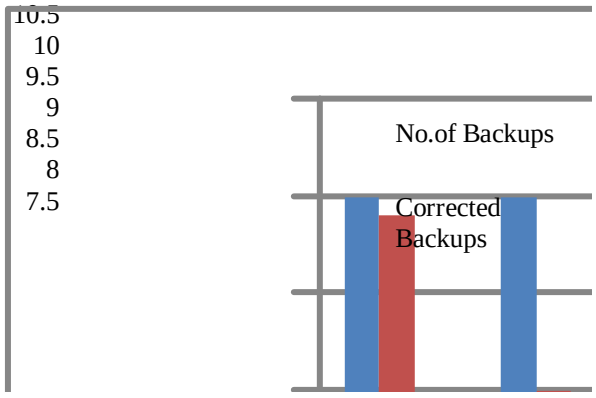

Figure 3: Performance of Fused Backups

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.

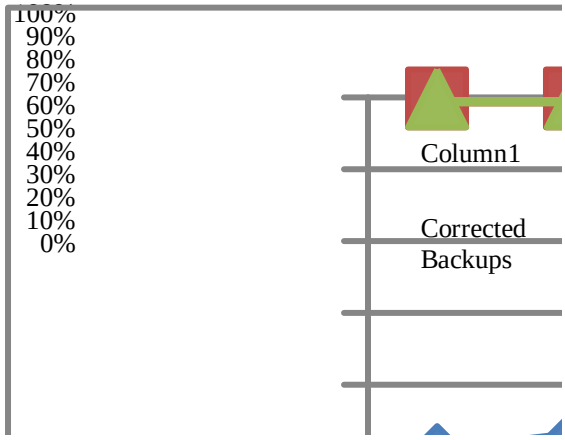

Figure 4: Time Complexity of Fused Backups

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

6. Conclusions and Future Work

A fusion-based technique for fault tolerance 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. The system performance is increasing when we transferring the bulk of 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] Vijay K. Garg. Implementing fault-tolerant services using state machines:Beyond replication. In DISC, pages 450–464, 2010.
[3] 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.
[4] 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.
[5] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. IEEE Trans. Parallel Distrib. Syst., 1(1):17–25, January 1990.
[6] J.S. Plank and L. Xu, "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.
[7] M.O. Rabin, "Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance," J. ACM, vol. 36, no. 2, pp. 335-348, 1989.
[8] I.S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," J. Soc. for Industrial and Applied Math., vol. 8, no. 2, pp. 300-304, 1960.
[9] F.B. Schneider, "Byzantine Generals in Action: Implementing Fail- Stop Processors," ACM Trans. Computer Systems, vol. 2, no. 2, pp. 145-154, 1984.
[10] F.B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," ACM Computing Surveys, vol. 22, no. 4, pp. 299-319, 1990.
[11] C.E. Shannon, "A Mathematical Theory of Communication," Bell Systems Technical J., vol. 27, pp. 379-423 and 623-656, 1948.
[12] J.K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazie`res, S. Mitra, A. Narayanan, M. Rosenblum, S.M. Rumble, E. Stratmann, and R. Stutsman, "The Case for RAMClouds: Scalable High-Performance Storage Entirely in Dram," ACM SIGOPS Operating Systems Rev., vol. 43, pp. 92-105, 2009.
[13] D.A. Patterson, G. Gibson, and R.H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '88), pp. 109-116, 1988.
[14] W.W. Peterson and E.J. Weldon, Error-Correcting Codes - Revised, second ed. The MIT Press, Mar. 1972.
[15] J.S. Plank, "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.