The Dissertation Committee for Bharath Balasubramanian
certifies that this is the approved version of the following dissertation:

Fault Tolerance in Distributed Systems: A Coding-Theoretic Approach

Committee:

Vijay K. Garg, Supervisor

Christine L. Julien

Craig M. Chase

Greg Plaxton

Nur A. Touba

Sriram Vishwanath

Fault Tolerance in Distributed Systems: A Coding-Theoretic Approach

by

Bharath Balasubramanian, B.E.; M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2012

To my parents, B. Sudha and R. Balasubramanian.

Acknowledgments

I wish to use this opportunity to thank the people who helped me during the course of my PhD. My advisor, Dr. Garg, had to face the challenge of supervising me while he was on a two year sabbatical. He was in full control of my PhD, from problem definition to my dissertation and future statement of teaching and research. The following are some of his qualities that I wish to emulate: (*i*) His keen appreciation of fundamental problems, immaterial of how easy or hard they look at first glance. (*ii*) A passion for concise elegant writing. For at least two of my papers, we have sat down together and he would read each and every line of important sections aloud, following which we would discuss if the line was framed right. (*iii*) The motivation to keep writing single author papers and Java code for a lot of his work, even at this level of seniority, all by himself. (*iv*) His pleasant zen-like manner, never once pushing more than is required. Even if nothing else goes right academically from here on, no one can take away the fact that Dr. Garg was more or less happy with my PhD. Despite the modern dictates of first-name addressal, he will always be 'Sir' to me.

Dr. Touba, Dr. Plaxton and Dr. Vishwanath helped shape this dissertation with their comments and suggestions during both the proposal and the defense. Dr. Christine has been generous with her advice at various points of my PhD. I am amazed at the thoroughness with which she has read and commented on this dissertation. Her presence on my committee gave me a sense of calm and support. Dr. Chase has been my biggest source of academic strength

after Dr. Garg. Doing well in Dr. Chase's class was the first time I genuinely felt I belonged at UT. Never has computer science looked more cooler than when he taught it. He had an influence on many aspects of this work. I also wish to thank my favourite teachers Arun Roy, Dr. Garg, Dr. Chase, Dr. Plaxton, Dr. De Viciana and Dr. Patt for shaping my fundamentals. Your passion for teaching is an inspiration.

RoseAnna has ensured that I never had to worry about the administrative side of the PhD, making sure that all payments and reimbursments went through with minimal effort from my side. Melanie can add one more to the list of ECE graduates she has nurtured and shepherded through school, always taking that additional step to make sure that everything is in place for us.

The conviction that I could do my PhD started with Vinit Ogale, who first taught me in my undergraduate college. He had a dominating influence in the courses that I took at UT, the lab that I joined and eventually the papers that I wrote. That apart, I have wonderful memories of our research sessions (80% gossip + 20% research). I thank the members of PDSLAB: Selma, John, Yen-Jung, Wei-Lun and Himanshu for their patience while both talking and listening to me.

It is with great fondness that I look upon my academic discussions with Aditya, Ayis, Bofi, Harish, Khubaib, Meghana, Np, Roopsha and Vasanth. Talking about my work with them helped clarify many a point. More importantly, it was gratifying to hear them discuss their work and solicit my opinions on it. And how can I forget the many afternoons spent around the ENS porch with these guys, discussing just about everything under the sun?

If I have completed this PhD, despite some upheaval, it is because of a tremendous support system. Amresh, Arun, Ashwin, Bhal, Bong, Manoj,

Mayank, Poorna, Rohan, Sangeetha, Shobha, Simone, Smita, Thiagu, Tinny, Uk and Vineet have been a constant source of material and emotional support. I am yet to face pain deep enough that it cannot be controlled by spending time with these friends. There is only so much one can brood, when there is so much fun to be had.

The RC family had an early influence on me by making math and science seem like fun. The RVG family, my quasi grandparents, have spoilt and pampered me to a fault. Karthik, my brother, still remains one of the most interesting people to talk to. In moments of laziness, his uncompromising attitude towards his passions, often servers as a check.

My late mother, has made many a sacrifice to ensure that our family functions well. That she is no longer around to see me graduate is a tragedy that will never leave me. Nevertheless, her qualities will live on through me and my brother. And of course, that most fault tolerant of finite state machines, my father. He encouraged me to continue my PhD at a time when all logic demanded that I stop it and stay with him. I hope I have inherited some of that bullish will to survive, flourish and improve.

Last and least, I wish to thank two auxiliary components that have kept me sane through my PhD. I wish to thank every one remotely associated with Indian cricket between the 2000 ICC knockout tournament to the crowning glory of the 2011 ICC world cup. Cricket's influence on my moods and thereby my work never ceases to baffle me. Finally, my beloved, overpriced, JP's Java coffee shop, deserves a doff of the hat for sometimes achieving what gleaming labs and brand new buildings could not: making the graduate student work.

Fault Tolerance in Distributed Systems: A Coding-Theoretic Approach

Publication No. ___________________

Bharath Balasubramanian, Ph.D.
The University of Texas at Austin, 2012

Supervisor: Vijay K. Garg

Distributed systems are rapidly increasing in importance due to the need for scalable computations on huge volumes of data. This fact is reflected in many real-world distributed applications such as Amazon's EC2 cloud computing service, Facebook's Cassandra key-value store or Apache's Hadoop MapReduce framework. Multi-core architectures developed by companies such as Intel and AMD have further brought this to prominence, since workloads can now be distributed across many individual cores. The nodes or entities in such systems are often built using commodity hardware and are prone to physical failures and security vulnerabilities. Achieving fault tolerance in such systems is a challenging task, since it is not easy to observe and control these distributed entities.

Replication is a standard approach for fault tolerance in distributed systems. The main advantage of this approach is that the backups incur very little overhead in terms of the time taken for normal operation or recovery.

However, replication is grossly wasteful in terms of the number of backups required for fault tolerance. The large number of backups has two major implications. First, the total space or memory required for fault tolerance is considerably high. Second, there is a significant cost of resources such as the power required to run the backup processes. Given the large number of distributed servers employed in real-world applications, it is a hard task to provide fault tolerance while achieving both space and operational efficiency.

In the world of data fault tolerance and communication, coding theory is used as the space efficient alternate for replication. A direct application of coding theory to distributed servers, treating the servers as blocks of data, is very inefficient in terms of the updates to the backups. This is primarily because each update to the server will affect many blocks in memory, all of which have to be re-encoded at the backups. This leads us to the following thesis statement: Can we design a mechanism for fault tolerance in distributed systems that combines the space efficiency of coding theory with the low operational overhead of replication?

We present a new paradigm to solve this problem, broadly referred to as *fusion*. We provide fusion-based solutions for two models of computation that are representative of a large class of applications: (*i*) Systems modeled as deterministic finite state machines and, (*ii*) Systems modeled as programs containing data structures. For finite state machines, we use the notion of Hamming distances to present a polynomial time algorithm to generate efficient backup state machines. For programs hosting data structures, we use a combination of erasure codes and selective replication to generate efficient backups for most commonly used data structures such as queues, array lists, linked lists, vectors and maps. We present theoretical and experimental results that demonstrate the efficiency of our schemes over replication. Finally, we use our schemes to design an efficient solution for fault tolerance in two real-world applications: Amazons Dynamo key-value store, and Google's Map Reduce framework.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The replicated state machine approach is a standard method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas [25, 51, 53, 82, 83, 86, 89, 91]. In this approach, to correct f crashes [82] of a primary server, one must have $f + 1$ identical copies of that server. If all $f + 1$ servers/processes start with identical state, are completely deterministic in execution, and agree on the set and the order of commands they execute, then their state will be identical at all times. This means that failure of f of them will leave at least one copy available.

To correct f Byzantine faults [53], where a faulty server can lie and behave maliciously, one must have $2f + 1$ copies of each server. This ensures that the majority among the copies is guaranteed to be correct. If we have n distinct servers in the system (referred to as *primaries*), replication requires nf additional backup servers for crash faults and $2nf$ additional backup servers for Byzantine faults.

Replication is prevalent in many real-world distributed applications. For example, it is used in map-reduce based fault-tolerant distributed computation [24] where different tasks are sent out to workers that run on machines that can fail. In these systems, a single task is farmed out to multiple workers to ensure that it is completed in spite of faults. As another example, consider a system of n robots (or sensors) performing different sets of tasks, or similar tasks on different data. To correct the failure of a single robot, say due to power loss or physical damage, the replication approach requires one additional copy of each of the robots. Common to these examples is the fact that computation is replicated to correct failures. Since we maintain a replica for each primary, this approach is expensive in terms of both the storage space consumed by the backups as well as the resources such as the power required to maintain them.

It must be noted that replication has been considered wasteful in the context of data fault-tolerance for many decades. In data storage and communication, coding theory [9, 57, 73] is extensively used to recover from faults. For example, to tolerate a single crash fault (or erasure) among two data items A and B, rather than keep a copy of both these items, we can just keep the XOR or sum of A and B. Either of the failed items can be recovered using the backup and the surviving data item. This is the basic premise of RAID disks that use disk striping and parity based schemes to recover from disk faults [18, 72, 74]. Erasure codes are used extensively in the area of distributed storage of data [16, 41]. Similarly, network coding [15, 60] has been used for recovering from packet loss or to reduce the communication overhead for multicast. In these applications, the data is viewed as a set of data entities such as disk blocks for storage applications and packets for network applications. These coding-theoretic techniques give much better space utilization than replication.

Each of the above two methods, replication and coding theory, have their advantages and disadvantages. Replication is expensive from the space perspective, especially when there are a large number of primary servers. However, backup maintenance and recovery from failure is simple for replication.

Coding theory is efficient in space or resource utilization but is rarely applied to backup active servers. This is mainly because, with each update, the system state changes considerably and re-encoding the backups corresponding to each of these updates is expensive both in terms of computation and communication. The natural scientific and practical question that arises is if we can get the best of both these worlds or at least a mechanism that is close to coding in resource utilization and close to replication in the efficiency of maintaining backups.

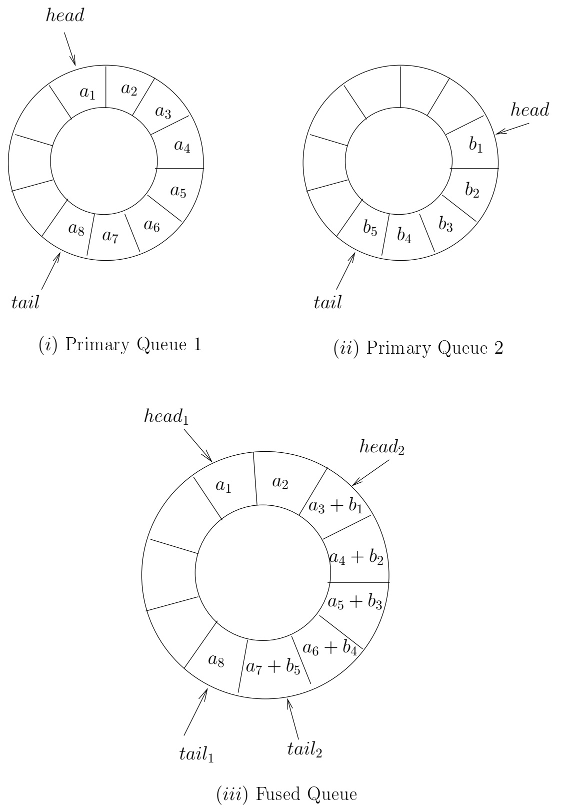
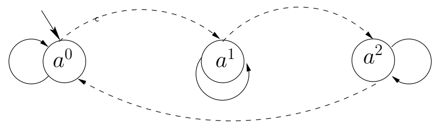


Figure 1.1: Lock Servers: Fused Circular Queue.

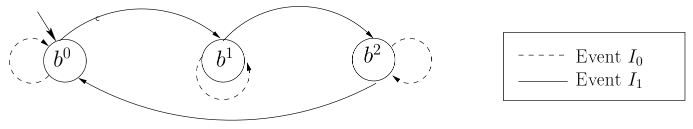
For example, consider a set of lock servers that maintain and coordinate the use of locks. Each lock server maintains a list of pending requests in the form of a queue. To correct even one crash fault among n such servers, replication requires n backups. A simple coding-theoretic solution to this problem is to treat the memory in these servers as blocks of striped data and maintain checksum blocks corresponding to them. Using appropriate codes, one crash fault can be corrected using just one additional checksum block. This is n times more space efficient than replication. However, when an update occurs at the lock server, all the stripes that have changed need to be sent to the backups. Since these servers are complex, a simple update will result in a change in many of the stripes, rendering the updates expensive.

In this dissertation, we propose an alternate method called *fusion* for fault-tolerance in distributed systems that is efficient in resource utilization as well as maintenance of backups. Consider the example of the primary lock server queues shown in Fig. 1.1. To correct one crash fault among these queues we can maintain a single backup server that contains a *fused* queue. Here, the nodes of the fused queue contain the primary elements in the coded form (in this case, simply sum). When an element is added to a primary queue, it is sent to the backup, which updates its queue accordingly. This solution is as space efficient as coding theory, and an update to the backup takes only as long as it does at the primary.

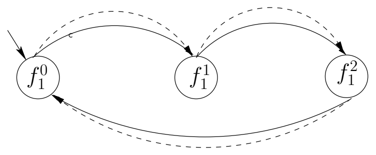
As another example of fusion, consider two sensors or robots counting two distinct events such as the number of people entering a room and the number of people leaving a room. In Fig. 1.2, sensor A is executing a deterministic finite state machine (referred to as DFSM or just machine) that counts the number of events I_0, modulo 3. Sensor B executes a machine that counts the number of events I_1, modulo 3. To correct one crash fault among these two machines i.e, to recover the state in which the machine was executing, repli-



(*i*) $A(I_0 \bmod 3 \text{ counter})$



(*ii*) $B(I_1 \bmod 3 \text{ counter})$



(*iii*) $F(\{I_0 + I_1\} \bmod 3 \text{ counter})$

Figure 1.2: Sensors with Event Counters: Fused Event Counter.

cation requires a copy of each of these machines. For this simple example, it is clear that we can correct one fault by maintaining just one additional *fused* machine F that counts the number of events $\{I_0 + I_1\}$, modulo 3, as shown in Fig. 1.2. If A, B and F start from their initial states and receive the same sequence of input events, then, given the state of any two machines, we can generate the state of the remaining machine.

In this case, fusion requires just one backup with three states as compared to replication that requires two backups each containing three states. Since the backups are independently executing machines, an update to the backup takes only as much time as the corresponding update to the primary. This brings us to some of the basic goals of fusion and the project:

- The case of lock servers or counters was easy. Can this be extended to computations involving complicated data structures or state machines?
- Is there an efficient way to detect faults in untrusted systems?
- How can we correct multiple crash and Byzantine faults?

There are three fundamental characteristics that distinguish our proposed notion of fusion from replication and coding as summarized in Table 1.1. The first difference is that when an operation is applied to primary data, in replication we apply the same operation to data at the backup, i.e., backups are operation-oriented and need no data from the primaries. For coding, we ignore the computation and simply focus on data. When data changes, the resulting code needs to be recomputed from the new data sent by the primary. Thus, coding theory is data-oriented. Fusion uses a combination of redundancy in computation and data to maintain the backups. The update to backup data may require some computation that is operation specific and some that is data specific depending upon the nature of the update.

Characteristic	**Replication**	**Coding**	**Fusion**
Update of Backups	Operation-Oriented	Data-Oriented	Combination
Treatment of Data	Complete replication	No replication	Selective replication
Operation Awareness	None	None	Aware

Table 1.1: Comparison of Replication, Coding and Fusion

The second difference is in the treatment of data. Both replication and coding treat all data items uniformly; one replicates all data, and the other replicates none. Fusion selectively replicates data so that updates to the backups can be carried out efficiently. Finally, note that both replication and coding theory are oblivious to the computation and the structure of primary data that we want to make fault-tolerant. For example, in coding theory the details of actual operations on the data are ignored, and the techniques simply recompute the encoded backups after any write update. Fusion, on the other hand, exploits the structure of the service that we want to make fault-tolerant. Thus, the fusion algorithm for queues may be different from the fusion algorithm for binary trees.

To make our techniques useful in a wide range of distributed applications, we explore fusion in two contexts: (i) distributed systems modeled as deterministic finite state machines [5, 7, 66] and (ii) distributed applications hosting large data structures [3, 6].

Given a set of n different primary DFSMs, we present a technique to correct f crash faults or $\lfloor f/2 \rfloor$ Byzantine faults using just f additional fused state machines. Further, we present a fusion algorithm that ensures that the backups are optimized for states as well as events. Our experiments on common benchmarks for DFSMs indicate that fusion achieves significant savings in space over replication (38% on average).

Given a set of n different primary data structures, we present a technique to correct f crash faults using just f additional fused data structures. We present a design of fused backups for most commonly used data structures such as lists, sets, maps and trees. The fused backups achieve $O(n)$ savings over replication while incurring no additional overhead for updates. Our experimental evaluation confirms that fusion achieves almost n times savings in space over replication, while it is only 1.5 times slower in terms of the update time. As expected, recovery is much faster in replication.

It is important to note that there are at least two advantages of running fewer backup processes. First, the amount of state and therefore the storage maintained at the backup servers is reduced. Second, by reducing the number of processes in the system, we save on the power and other resources required to run the backup servers. However, there are also disadvantages to this approach. Since the data elements at the backup process are maintained in the fused/coded form, recovery is expensive. The other significant disadvantage of fusion is the increased load that needs to be serviced at the backup processes. As each backup contains the data elements belonging to all the primaries, it also needs to receive updates corresponding to all of them. In many practical scenarios, the backup may not have the necessary bandwidth or resources to service these many requests, especially when n is large.

In the future, we wish to explore techniques that offer different trade-offs between space, load and recovery cost. For example, rather than maintaining backups that contain elements from all primaries, we can partition the set of primaries and maintain backups for these smaller blocks. In the following sections, we summarize our fusion-based techniques for state machines and data structures.

1.1 Fused State Machines

Distributed applications often use DFSMs to model computations such as regular expressions for pattern detection, syntactical analysis of documents or mining algorithms for large data sets [43, 54, 65]. DFSMs are also used in many other areas such as digital hardware systems, network protocol specification and sensor networks [12, 28, 40, 42, 46, 94]. In this section, we present a fusion-based solution for fault tolerance in DFSMs that is much more space efficient than replication.

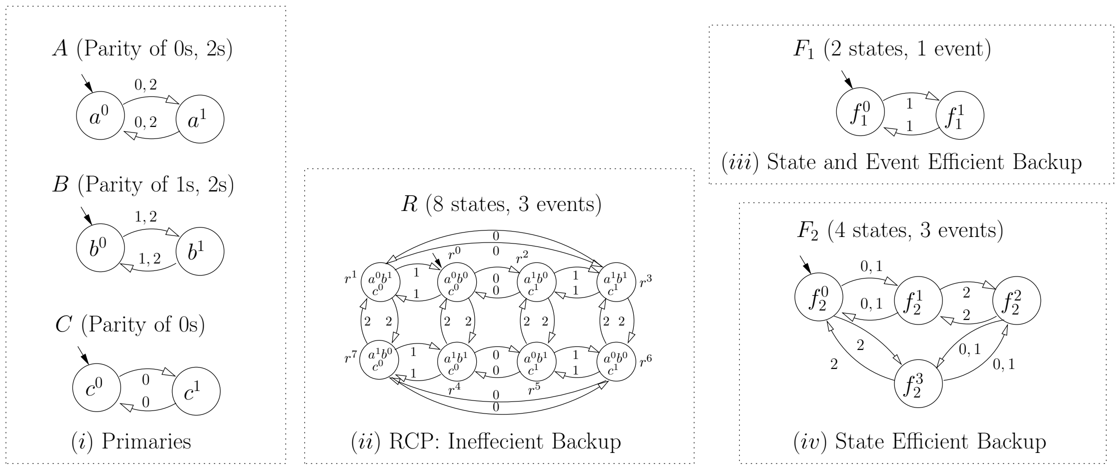


Figure 1.3: Correcting one crash fault among $\{A, B, C\}$ using just one backup.

Consider a distributed application that is searching for three different string patterns in a file, as modeled by the three parity-check machines A, B and C shown in Fig. 1.3. In this example, A checks for the parity of $\{0+2\}$s in the input file. So if the input sequence seen by A is $2 \rightarrow 0 \rightarrow 2$, then starting from the initial state a^0, A's transitions are: $a^0 \rightarrow a^1 \rightarrow a^0 \rightarrow a^1$. The state of machine A after acting on the input sequence, i.e., state a^1, confirms that there is an odd number of $\{0+2\}$s in the input sequence. Machines B and C check for the parity of $\{1+2\}$s and $\{0\}$s respectively.

Another way of looking at replication in DFSMs is to construct a backup machine that is the *reachable cross product* or *RCP* (defined formally in section 4.1.1) of the original machines. The reachable cross product R of the primaries A, B and C is shown in Fig. 1.3. Each state of R is a tuple in which the elements correspond to the states of A, B and C respectively. Let each of the machines A, B, C and R start from their initial state. If some input sequence $0 \rightarrow 2 \rightarrow 1$ is applied on these machines, then the states of R, A, B and C are $r^6 = \{a^0 b^0 c^1\}$, a^0, b^0 and c^1 respectively. Here, even if one of the primaries crash, using the state of R, we can determine the state of the crashed primary. Hence, the RCP is a valid backup machine.

However, using the RCP as a backup has two major disadvantages: (i) Given n primaries each containing $O(s)$ states, the number of states in the RCP is $O(s^n)$, which is exponential in the number of primaries. In Fig. 1.3, since each primary has two states, R has eight states. (ii) The event set of the RCP is the union of the event sets of the primaries. In Fig. 1.3 while A and B have two events in their event set, R has three events. This translates to increased load on the backup. Can we generate backup machines that are more efficient than the RCP in terms of states and events?

Consider machine F_1 shown in Fig. 1.3. If an input sequence $0 \rightarrow 0 \rightarrow 1 \rightarrow 2$ is applied on the machines A, B, C and F_1, then they will be in states a^1, b^0, c^0 and f_1^1. Assume a crash fault in C. Given the parity of 1s (state of F_1) and the parity of $\{1+2\}$s (state of B), we can first determine the parity of 2s. Using this, and the parity of $\{0+2\}$s (state of A), we can determine the parity of 0s (state of C). Hence, we can determine the state of C as c^0 using the states of A, B and F_1. This argument can be extended to correcting one fault among any of the machines in $\{A, B, C, F_1\}$.

Fault tolerance using F_1 consumes fewer backups than replication (one vs. three), fewer states than the RCP (two states vs. eight states) and fewer events than the RCP (one event vs. three events). How can we generate such a backup for any arbitrary set of machines? In Fig. 1.3, can F_1 and F_2 can correct two crash faults among the primaries? Further, how do we correct the faults? In this dissertation, we address such questions through the following contributions.

1.1.1 Contributions

Framework for Fault Tolerance in DFSMs We develop a framework to understand fault tolerance in DFSMs based on the idea of a *fault graph* and Hamming distances [38] for a set of machines. Using this framework, we can specify the exact number of crash or Byzantine faults a set of machines can correct. Further, we introduce the concept of an *(f, m)-fusion*, which is a set of m machines that can correct f crash faults, detect f Byzantine faults or correct $\lfloor f/2 \rfloor$ Byzantine faults. We refer to the machines as *fusions* or *fused backups*. It can be shown that machines F_1 and F_2 in Fig. 1.3 can correct two crash faults among $\{A, B, C\}$. Hence $\{F_1, F_2\}$ is a (2, 2)-fusion of $\{A, B, C\}$. Replication is just a special case of $(f,\ m)$-fusion where $m = nf$. We prove properties on the $(f,\ m)$-fusion for a given set of machines including lower bounds for the existence of such fusions.

Algorithm to Generate Fused Backup Machines Given a set of n primaries we present an algorithm that generates an $(f,\ f)$-fusion corresponding to them, i.e., we generate a set of f backup machines that can correct f crash or $\lfloor f/2 \rfloor$ Byzantine faults among them. Note that, replication requires nf backups to correct the same number of faults. We show that our backups are efficient in terms of: (i) number of states in each backup (ii) number of events in each backup, and (iii) *minimality* of the entire set of backups in terms of states. Further, we show that if our algorithm does not achieve state and event reduction, then no solution with the same number of backups achieves it. Our algorithm has time complexity polynomial in N, where N is the number of states in the RCP of the primaries. We present an incremental approach to this algorithm that improves the time complexity by a factor of $O(\rho^n)$, where

ρ is the average state savings achieved by fusion.

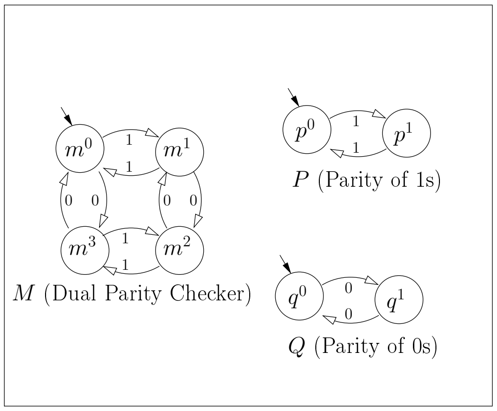


Figure 1.4: Event-based decomposition of a machine.

Event-based Decomposition of DFSMs We pose a question that is fundamental to the understanding of DFSMs, independent of fault-tolerance: Given a machine M, can it be *replaced* by two or more machines executing in parallel, each containing fewer events than M? In other words, given the state of these fewer-event machines, can we uniquely determine the state of M? In Fig. 1.4, the 2-event machine M (it contains events 0 and 1 in its event set), checks for the parity of 0s *and* 1s. M can be replaced by two 1-event machines P and Q, that check for the parity of just 1s or 0s respectively. Given the state of P and Q, we can determine the state of M. How can we generate these event-reduced machines (if they exist) for any given machine? While there has been work on both the state-based decomposition [39, 55] and the minimization of completely specified machines [44, 45], this is the first work that addresses the problem of event-reduction.

In this dissertation, we define the concept of a (k,e)-event decomposition of a machine M that is a set of k machines, each with at least e events fewer than the event set of M, such that given the state of these machines, we can determine the state of M. We present an algorithm to generate such machines with time complexity $O(|X_M||\Sigma_M|^{e+1} + |X_M|^3|\Sigma_M|^e)$, where X_M is the set of states and Σ_M is the set of events of M. The load on a process running a machine is directly proportional to the number of events in the event-set of the machine. Hence, this decomposition is crucial for applications such as sensor networks in which there are strict limits on the number of events that each process can service.

Detection and Correction of Faults We present a Byzantine detection algorithm with time complexity $O(nf)$ on average, which is the same as the time complexity of detection for replication. Hence, for a system that needs to periodically detect liars, fusion causes no additional overhead. We reduce the problem of fault correction to one of finding points within a certain Hamming distance of a given query point in n-dimensional space and present algorithms to correct crash and Byzantine faults with time complexity $O(n\rho f)$ with high probability (w.h.p). The time complexities for crash and Byzantine correction in replication are $O(f)$ and $O(nf)$ respectively. Hence, for small values of n and ρ, fusion causes almost no overhead for recovery.

Fusion-based Grep in the MapReduce Framework To illustrate the practical use of fusion, we apply its design to the *grep* application of the MapReduce framework [24]. Currently, many large scale distributed applications are built using the MapReduce framework. The grep functionality is used in many applications such as data mining, machine learning, and query log analysis, that need to identify patterns in large volumes of textual data.

Using a simple example, we show that a pure replication-based approach for fault tolerance needs 1.8 million map tasks while our fusion-based solution requires only 1.4 million map tasks. Further, we show that our approach causes minimal overhead during normal operation or recovery.

Fusion-based Design Tool and Experimental Evaluation We provide a Java design tool [4], based on our fusion algorithm, that takes a set of input machines and generates fused backup machines corresponding to them. We evaluate our fusion algorithm on the MCNC'91 [95] benchmarks for DFSMs, that are widely used in the fields of logic synthesis and circuit design. Our results show that the average state space savings in fusion (over replication) is 38% (range 0-99%), while the average event-reduction is 4% (range 0-45%). Further, the average savings in time by the incremental approach for generating the fusions (over the non-incremental approach) is 8%. In the following section, we present our fusion-based approach for fault tolerance in data structures.

1.2 Fused Data Structures

Servers in a distributed system often maintain large instance of data structures to store and manipulate data. Examples include Amazon's *Dynamo* or Facebook's *Cassandra* key-value store, the *OceanStore* storage service or Google's *Chubby* lock service [13, 25, 48, 49]. In this section, we present a technique for fault tolerance in data structures using fused backups that combines the space efficiency of coding and the minimal update overhead of replication. The fused backups maintain primary data in the coded form to save space, while they replicate the index structure of each primary to enable efficient updates.

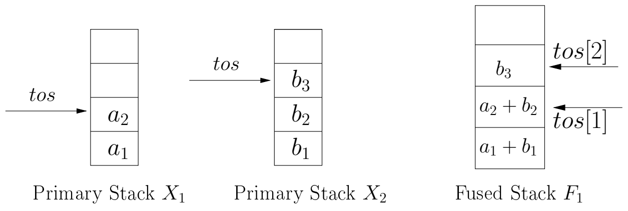


Figure 1.5: Correcting one crash fault among X_1 and X_2 using just one backup.

In Fig. 1.5, 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 pushed on to X_1, this element is sent to the fused stack and the value of the second node (counting from zero) is updated to $a_3 + b_3$. Similarly, in the case of a pop to X_2, of say b_3, the second node of F_1 is updated to a_3.

The set of data structures in Fig. 1.5 can tolerate 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. The savings in space is achieved by fusing the data nodes, while the index structure at the backups allows for efficient updates. Note that, to correct one crash fault, the fusion-based solution just requires one backup with three nodes as compared to replication which requires two backups each containing three nodes. How can we correct faults among other data structures such as tree-maps and hash tables? Can we ensure that each backup has a minimum number of nodes? How do we perform recovery efficiently? We answer such questions through the following contributions.

1.2.1 Contributions

Fusion-based Fault Tolerant Data Structures We present a technique for fault tolerance in data structures using fused backups. Using our technique, we can correct f crash faults among n instances of most commonly used data structures such as stacks, vectors, binary search trees, hash maps and hash tables using just f additional backups. Note that, replication requires nf backups to correct the same number of faults. In Fig. 1.5, 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. These set of data structures can tolerate two crash faults. We extend this idea for values of f greater than two using Reed-Solomon (RS) erasure codes [11, 76, 79], which are widely used to generate the optimal number of parity blocks in RAID-like systems.

Detection and Correction of Faults We present algorithms for the detection and correction of faults among the primary data structures and the fused backups. Crash faults in a synchronous system, such as the one assumed in our model, can easily be detected using time outs. To detect Byzantine faults, the state of the data structures needs to be inspected on every update to ensure that there are no liars in the system. In this dissertation, we present a solution to correct f Byzantine faults among n primary data structures using just $nf + f$ backup structures as compared to the $2nf$ backups required by replication. We use a combination of replication and fusion to ensure minimal overhead during normal operation.

Theory of Fused Data Structures We prove properties on our fused backups such as space optimality, update efficiency and order independence. Given n primaries, our approach achieves $O(n)$ times savings in space over replication. The time complexity of updates to our backups is identical to that for replication, and the updates to the backups can be done with a high level of concurrency. Further, we show that the updates to different backups from distinct primaries can be received in any order, thereby eliminating the need for synchronization at the backups.

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. Consider a set of n data structures, each residing on a distinct server. We need to tolerate f crash faults among the servers given only γ additional servers to host the backup structures. We present a solution to this problem that requires $\lceil n/(n+\gamma-f) \rceil f$ backups and show that this is the necessary and sufficient number of backups for this problem. We also present a way to compare (or order) sets of backups of the same size, based on the number of primaries that they need to service. This is an important parameter because the load on a backup is directly proportional to the number of primaries it has to service. We show that our partitioning algorithm generates a *minimal* set of backups.

Fusion-based Key-Value Store To illustrate the practical usefulness of fusion, we apply our design to Amazon's Dynamo [25], which is the data-store underlying many of the services provided by Amazon. 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 fusion and replication, which requires far fewer backups, while providing almost the same levels of durability and availability for writes. In a typical host cluster, where there are 100 dynamo hosts each hosting a data structure, the current replication-based approach requires 300 backup structures. Our approach, on the other hand, requires only 120 backup structures. This translates to significant savings in both the space occupied by the backups as well as the infrastructure costs such as the power required to run these backups.

Fused Data Structure Library and Experimental Evaluation We provide a Java library/package of fused backups [2] for all the data structures in the Java Collection Framework. This includes most commonly used data structures such as lists, maps, vectors, and stacks. We also performed experiments comparing the performance of the fused data structures with replication. The results indicate that the current version of fusion is very space efficient as compared to replication (approximately n times) while the time taken to update the backups is only marginally more (approximately 1.5 times slower). Though the time taken for recovery is far more in fusion as compared to replication, in absolute terms, it is still low enough to be practical (order of milliseconds).

In the following section, we describe our system model and assumptions, applicable to both fused data structures and fused state machines.

1.3 System Model and Assumptions

The model used in our work is based on the model presented in Fred Schneider's tutorial on the replicated state machine approach [83]. Our system consists of a set of distinct distributed *processes* or *servers* with no shared memory among them. Each process defines a *state machine* that consists of (*i*) a set of states, (*ii*) a set of atomic actions each of which deterministically transitions the state machine from one state to another (*iii*) an output associated with each state transition. The *clients* of these processes send requests to the processes with the action that they need to perform. Further, the clients receive and act on the output received from the processes. These are the main assumptions we make on our system:

- The processes can undergo two types of faults or failures:

    1. *Crash* or *Fail-stop* faults in which there is a loss in the state of the process and/or the process stops responding. This kind of failure can be detected by other processes.

    2. *Byzantine* faults in which the process can behave in an arbitrarily malicious manner and even collude with other processes to foil any protocol. However, they cannot corrupt their identity. Byzantine faults are the worst kind of faults any system can face. The requirement on identity is easily achieved by maintaining the cryptographic hash of the identity of each process at every other process.

- The given set of processes or state machines in our system cannot detect or correct a single fault just by themselves. Hence, we need to add backup processes to make the system fault tolerant.

- Our system is fully connected, and the clients and the processes can send messages to each other. The communication channels are loss-less and guarantee the first-in-first-out property (FIFO). In other words, if a process A sends message m_1 followed by message m_2 to some other process B, then B receives m_1 followed by m_2. Both these requirements are guaranteed by implementing the TCP protocol for communication.

- There is a strict upper bound on the time taken for all actions in our system (including message delivery), i.e., we assume a synchronous system. Note that building fault-tolerant systems in the presence of even crash faults is impossible in fully asynchronous systems [30].

- The clients in our system are fault-free. It may be possible to consider the client as one of the processes defining a state machine and build a fault tolerant system in which the clients fail [83]. However, for most practical systems it is far more relevant to reason about the correctness of the system assuming that the clients are fault-free.

1.4 Overview of the Dissertation

The rest of this dissertation is organized as follows. In chapter 2, we compare fusion with other related areas of work. In chapter 3, we present some of the standard concepts in the literature that we use in our work. In chapters 4 and 5 we present our fusion-based solutions for fault tolerance in state machines and data structures respectively. Chapter 6 deals with the practical applications of our work. Finally, we conclude the dissertation in chapter 7 and present future directions in which this work can be extended.

Chapter 2

Related Work

In this chapter we discuss the main areas of work related to the topic of this dissertation. In the first two sections, we describe the two major topics of direct comparison to our work: replication and coding theory. In the final section, we discuss other associated areas such as rollback recovery protocols and finite state machine minimization.

2.1 Replication

Replication [25, 51, 53, 82, 83, 86, 89, 91] is the prevalent solution for fault tolerance in distributed systems. To tolerate f crash faults among n primary processes (or servers) in a distributed system, replication maintains $f + 1$ replicas of each process, resulting in a total of nf backups. These replicas can also tolerate $\lfloor f/2 \rfloor$ Byzantine faults since there is always a majority of correct copies available for each server.

Replication has many advantages. The solution is easy to design since the backups are identical to the primaries. The algorithms for the maintenance and recovery of the backups are straightforward. Each backup has to serve only as many requests as the corresponding primary, so the backups will be loaded only as much as the primary. Hence, there is very little overhead for normal operation. Also, recovery in replication is cheap in terms of time complexity and message complexity.

However, if we have n independent servers in the system, the replicated state machine approach results in nf backups for f crash faults or $\lfloor f/2 \rfloor$ Byzantine faults. This is extremely wasteful in terms of space and other infrastructure resources such as the power required to run these backups. In the following paragraphs we present an overview of the differences between replication and our fusion-based solutions for state machines and data structures. A detailed comparison of the various parameters are provided in sections 4.5 and 5.5.

Given n different DFSMs, we present a fusion-based solution that requires only f backups to correct f crash or $\lfloor f/2 \rfloor$ Byzantine faults as compared to the nf backups required by replication. Similar to replication, the backup state machines transition independently on events/inputs applied by the client, without any communication with the other machines. While in the worst case, our backup machines require as much state space as replication, our experimental results in section 6.3 confirm that for most examples, we achieve significant state space reduction. Our algorithm for the detection and correction of faults has almost the same time complexity as that for replication.

The main disadvantages of our fusion-based solution for DFSMs over replication are: (i) Our algorithm for generating the backup machines has exponential time complexity in n, where n is the number of primary machines. In replication, we just have to generate copies of the backups, and so, the time complexity for the generation of the backups is linear in n. However, the backups have to be generated only once and hence, this might be an acceptable cost for fusion. (ii) The event set of each fused backup is the union of the event sets of the primaries. This results in increased load on each backup. In section

4.2 we discuss a technique to reduce the number of events in the event set of each backup. Our results show that for many examples, the number of events can be reduced.

Given n different data structures, we present a fusion-based solution that requires only f backups to correct f crash faults as compared to the nf backups required by replication. We show that our backups achieve $O(n)$ savings in state space over replication. Further, we detect and correct f Byzantine faults using just $nf + f$ backups as compared to the $2nf$ backups required by replication. The main intuition behind our solution is to combine ideas from both coding and replication. In our fused data structures, the savings in state space is obtained by maintaining the data in the coded form, while the update efficiency is achieved because the index information is being replicated. Further, for the efficient detection and correction of Byzantine faults, we explicitly maintain replicas of each primary.

Our work in fused data structures has the following disadvantages over replication: (i) Similar to the case of state machines, each fused backup data structure has to service requests corresponding to all the primaries. (ii) In our current design of fused backups, we use the Reed Solomon erasure codes [79], whose recovery time complexity is $O(nf)$ times more expensive than replication. Note that both these disadvantages can be alleviated by fusing a smaller number of primaries, thereby compromising on the savings in space to improve the recovery time and the backup load. Also, other codes can be used that offer better trade-offs for these parameters [9, 73].

We make two observation regarding the terminology in this dissertation for both state machines and data structures. First, in replication, the replicas or backups for each server are identical. Hence, in most of the literature, replication is said to maintain '$f + 1$ replicas' to correct f crash faults. In fusion, the backups are not identical to the primary servers. For consistency in terminology, we say that replication requires f additional copies of each primary while fusion requires f additional fused backups for the entire set of primaries. Second, since replication is a fault-masking technique, in the literature, replication is said to 'tolerate' faults. However, for fusion, we need to decode the values and correct the faults in the system. In this dissertation we use the terms 'tolerate' and 'correct' interchangeably for both replication and fusion.

2.2 Coding Theory

Coding theory [9, 73] has been used as a space-efficient alternative to replication in the fields of communication and data storage. Data that needs to be transmitted across a channel is encoded using redundant bits that can correct errors introduced by a noisy channel [31, 58, 61, 85]. Network coding is a paradigm in which coding theory is used to improve the throughput while transmitting data across a network [15, 27, 60]. Applications of coding theory in the storage domain include RAID disks [18, 72, 74] for persistent storage or information dispersal algorithms (IDA) for fault tolerance in a set of data blocks [77]. Recently, the use of erasure codes for the distributed storage of data has gained prominence [16, 41, 56, 67, 93]. In these approaches to fault tolerance, data is divided into smaller chunks and a space efficient error/erasure code is used to store the data.

These storage techniques, though very space efficient, are primarily used to backup data that is maintained on disks. Real-world distributed systems often export services that have strict deadlines to meet. In such cases, data is rarely maintained on disks due to their slow access times. The active data structures in such systems are usually maintained in main memory or RAM. In fact, a recent proposal of 'RAMClouds' [68] suggests that online storage of data must be held in a distributed RAM, to enable fast access. In these cases, a direct use of coding-theoretic solutions, that are oblivious to the exact structure of the applications, is often wasteful. In the following paragraphs we compare our fusion-based solution with a direct coding-theoretic solution for fault tolerance in state machines and data structures.

In a direct coding-theoretic approach to fault tolerance in DFSMs, we have to maintain a distinct parity server that maintains the checksums for the states of the primaries. If we wish to maintain exactly f backups, then these checksums need to be some MDS code (explained in section 3.2) such as Reed Solomon codes. When the state of a primary changes, this change in state has to be communicated to the checksum servers, which can then locally update their codes. This direct coding-theoretic approach will guarantee $O(n)$ savings in state space over replication. However, this technique has two major disadvantages: (i) If each DFSM has $O(s)$ states then $O(\log s)$ bits need to be communicated to the checksum devices with every state change. (ii) In any MDS code such as Reed Solomon codes, recovery time complexity is $O(nf)$ times more than replication.

In our work on fused state machines, we use the notion of Hamming distances to design backup finite state machines that act independently on the events applied by the client or the environment. So there is no communication necessary between the primaries and the backups. Further, we show that our algorithms for the detection and correction of faults cause very little overhead over replication. In the worst case, the state space required by our approach can be much worse than this direct-coding theoretic approach. However, the experimental results indicate that for many examples, we do achieve significant savings in state space.

We now consider the case of distributed programs hosting data structures in main memory. In a simple coding-theoretic solution, we have to encode the memory blocks occupied by the data structures. Since a data structure is rarely maintained contiguously in main memory, a structure-oblivious solution will have to encode all memory blocks that are associated with the implementation of this data structure in main memory. This approach has the following disadvantages: (*i*) it is not space efficient, since there could be a large number of such blocks in the form of free lists and memory book keeping information, (*ii*) 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 and, (*iii*) each operating system manages the heap in a different way and the solution to code the blocks is not operating system independent.

In our work on fused data structures, we present a design of backup data structures that is based on the design of the primary data structures. We use erasure codes, but only to encode the data elements of the primary and not the entire heap associated with the data structure. Since our solution is at the abstraction of the data structure, we are not concerned with how the operating system manages these structures in main memory. We also ensure that the communication and computation overhead for normal operation is similar to replication. The one disadvantage with our approach as compared to a structure-oblivious approach is that our design varies according to the primary data structure. So we have a different solution for linked lists and a different one for tree maps and so on. We alleviate this by providing a generic design which can be easily extended for all types of data structures. Further, we provide a library of such data structures for all the containers in the Java Collection Framework. Hence, users of the software are shielded from the complexity of design.

We also wish to point out another major difference between our use of coding theory and the standard usage in erasure-coded storage solutions. In distributed storage using erasure codes, a data element of a distributed process is split into many smaller chunks and then encoded. In our solution, we never split the data elements and instead *fuse* data elements of *different* processes to achieve savings in space. This difference is mainly because, unlike storage solutions, our approach is mainly designed for active systems. In such systems, data often needs to be read. For example, a data item could be a shopping cart entry in a key-value store. If this is data item is split, then to respond to a client read request, in the standard erasure-coded solutions we need to reconstruct the data block.

For example, assume two processes in the system, denoted by A and B respectively. In erasure-coded stores, to correct one fault, a data element a belonging to A is split into a_1 and a_2 and three blocks: a_1, a_2 and $a_1(XOR)a_2$ are maintained on different servers. In fusion, however, if the elements a and b belong to two primary data structures A and B respectively, we maintain $a(XOR)b$ in the fused data structure. In the former solution, to read the state of a, we need to acquire a_1 and a_2. In the fusion-based solution a is still available for efficient reads.

2.3 Other Areas of Work

Rollback Recovery Protocols Many distributed systems use checkpointing and rollback protocols for fault tolerance [21, 24, 29]. The basis of these approaches is that most processes in a distributed system have access to some form of stable storage such as hard disks. In *checkpointing* based approaches, the processes regularly log or write their state to stable storage, while in *logging* based approaches, processes log both their state and the non deterministic events that occur during execution. When a fault occurs, the state of the process can be recovered from storage and the process can be restarted or rolled back from that state. This approach requires no additional processes for fault tolerance. Also, recovery is simple and cheap in terms of time complexity.

The major disadvantage of this approach is that every time the process changes state, the new state has to be written to the stable storage/hard disk and disk access times are usually very high. So these approaches increase the time taken for fault-free normal operation. This is unacceptable for most application which have deadlines. Our fusion-based solution has only as much overhead as replication in terms of the time taken for normal operation.

Monitor Machines In digital systems, concurrent error detection techniques are often used to detect errors in the DFSMs of the system [28, 62]. The key idea behind this technique is to maintain one auxiliary machine for each primary DFSM. The auxiliary machine is designed in such a way that an error can be detected in the primary machine execution for each state transition. The main goal is to ensure that the auxiliary machine is as small as possible, so that it consumes lesser hardware than a simple replica of the primary machine.

A common way to design these auxiliary machines is to generate a machine homomorphic (explained in section 3.1) to the primary machine, by partitioning the states of the primary machine to create a smaller machine [70, 71]. This homomorphic machine, also referred to as a *monitor machine*, executes in lock-step with the corresponding primary machine. For any fault model, if the true and faulty successors of each state of the primary machine are in different blocks of the partition of the monitor machine, then an error can be detected. While we too generate machines homomorphic to the reachable cross product of the primary machines, our work is different from the work on monitor machines in the following ways: (*i*) Monitor machines focus on error detection in just one primary DFSM, while we focus on faults among a set of independent DFSMs (*ii*) The concurrent error detection techniques are only for error detection, while in our work, we handle the detection and correction of crash faults (similar to erasures) and Byzantine faults (similar to errors). (*iii*) In the fault model of monitor machines, each primary machine can enter only a small set of faulty states. This allows the reduction in the states of the monitor machine. In our work, the machines can enter any arbitrary state. We achieve reduction through the fact that, in the case of faults, recovery can be performed using information from the remaining machines.

DFSM Minimization Extensive work has been done [44, 45] on the minimization of completely specified DFSMs. In these approaches, the basic idea is to create equivalence classes of the state space of the DFSM and then combine them based on the transition functions. Even though we focus on reducing the reachable cross product of a given set of machines, it is important to note that the machines we generate may not be equivalent to the combined DFSM. Also, we implicitly assume that the input machines to our algorithm are reduced a priori using these minimization techniques.

Chapter 3

Background

In this chapter we present some of the standard concepts and results in the literature that are fundamental to our work. First, we describe the concept of machine decomposition that is relevant for our work on fused state machines. Second, we introduce the basics of linear coding with a focus on Reed-Solomon codes, which is an important part of fused data structures.

3.1 Machine Decomposition

Given a machine, we can partition its state space to generate other machines [39, 55]. Further, we can define a partial order over this set of machines, which in fact, forms a special structure called a lattice. In the following section, we define the concept of a lattice. Readers are referred to [10, 23] for a thorough treatment. Later, we describe the closed partition lattice of a given machine.

3.1.1 Lattices

A *relation* R over any set X is a subset of $X \times X$. A relation R is *reflexive* if for each $x \in X$, $(x, x) \in R$. A relation R is *antisymmetric* if for all $x, y \in X, \{(x, y) \in R \text{ and } (y, x) \in R\}$ implies, $x = y$. Finally, a relation R is transitive if for all $x, y, z \in X$, $\{(x, y) \in R \text{ and } (y, z) \in R\}$ implies $(x, z) \in R$.

For example, if X is the set of natural numbers N, then $R \equiv$ "*x divides y*" is a reflexive, antisymmetric and transitive relation over N. Clearly, $(2,4) \in R$ while $(3,7) \notin R$.

A reflexive partial order (or simply *partial order*) over a set is any relation that is reflexive, antisymmetric and transitive. The partial order is usually denoted by "$\leq$". A set X, along with a partial order $\leq$, on its elements, is denoted by $\langle X, \leq \rangle$ and is called a partially ordered set or a *poset*.

Definition 1. (Join and Meet of two elements) Let $a, b \in X$ where $\langle X, \leq \rangle$ is a poset.

For any element $c \in X$, we say that c is the join of a and b, i.e., $c = a \sqcup b$ iff,

1. $a \leq c$ and $b \leq c$
2. $\forall c' \in X, (a \leq c' \wedge b \leq c') \Rightarrow c \leq c'$.

For any $c \in X$, we say that c is the meet of a and b, i.e., $c = a \sqcap b$ iff,

1. $c \leq a$ and $c \leq b$
2. $\forall c' \in X, (c' \leq a \wedge c' \leq b) \Rightarrow c' \leq c$.

Note that the join and meet are also referred to as the *lowest upper bound* (LUB) and the *greatest lower bound* (GLB) respectively. Consider the poset $\langle N, \leq \rangle$, where N is the set of natural numbers and $\leq \equiv$ "*x divides y*". For any two elements in N, clearly the join is their lowest common multiple (LCM) and the meet is their highest common factor (HCF). Such a poset in which both the meet and join exist for all pairs of elements is called a *lattice*.

Definition 2. (Lattice) A poset $\langle X, \leq \rangle$ is a lattice iff $\forall a, b \in X$, $a \sqcup b \in X$ and $a \sqcap b \in X$.

We now define two special types of lattices, which for many algorithms on lattices, allow efficient computation.

Definition 3. (Distributive Lattice) A poset $\langle X, \leq \rangle$ is a distributive lattice iff $\forall a, b, c \in X : a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c)$

Definition 4. (Modular Lattice) A poset $\langle X, \leq \rangle$ is a modular lattice iff $\forall a, b, c \in X : a \geq c \Rightarrow a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup c$.

In the following section, we explain the concept of machine decomposition and the closed partition set of a given machine, based on the description in [55].

3.1.2 Closed Partition Set

Definition 5. A deterministic finite state machine (DFSM), denoted by A, is a quadruple, $(X_A, \Sigma_A, \alpha_A, a^0)$, where,

- X_A is the finite set of states corresponding to A.
- Σ_A is the finite set of events corresponding to A.
- $\alpha_A : X_A \times \Sigma_A \rightarrow X_A$, is the transition function corresponding to A. If the current state of A is s, and an event $\sigma \in \Sigma_A$ is applied on it, the next state can be uniquely determined as $\alpha_A(s, \sigma)$.
- a^0 is the initial state corresponding to A.

A *partition* P, on the state set X_A of a machine A is the set $\{B_1, \ldots, B_k\}$, of disjoint subsets of X_A, such that $\bigcup_{i=1}^{k} B_i = X_A$ and $B_i \cap B_j = \phi$ for $i \neq j$.

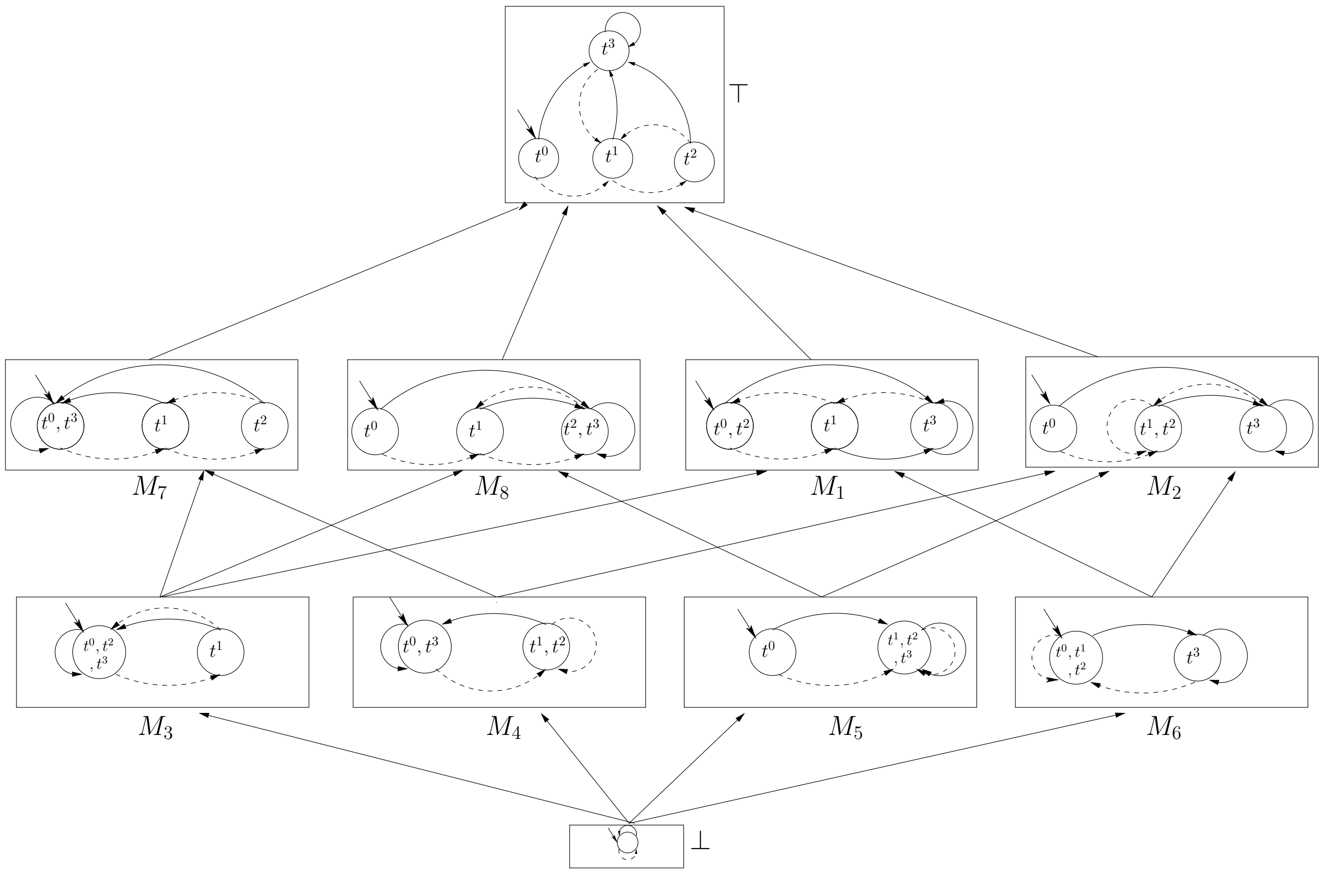


Figure 3.1: Closed partition set of $\top$.

An element B_i of a partition is called a *block*. If two states s and t are in the same block, we denote $s \equiv t(P)$. A partition, P, is said to be *closed* if each event, $\sigma \in \Sigma_A$, maps a block of P into another block of P. Formally, P is closed if for all $\sigma \in \Sigma_A$ and $s \equiv t(P)$, $\alpha_A(s, \sigma) \equiv \alpha_A(t, \sigma)$.

A closed partition P, corresponds to a distinct machine. For example, consider machine $\top$ shown in Fig. 3.1. Machine M_1 corresponds to a closed partition of the states of $\top$. The blocks of the partition, $\{t^0, t^2\}$, $\{t^1\}$ and $\{t^3\}$, correspond to the states of M_1. The closed partitions described here are also referred to as substitution property partitions or SP partitions in other literature [39]. We now define a partial order among machines.

Definition 6. ($\leq$ relation among machines) Given a machine A, consider two machines P_1 and P_2 corresponding to two closed partitions of X_A. Machine P_1 is less than or equal to P_2 ($P_1 \leq P_2$) if each block of P_2 is contained in a block of P_1.

P_1 is incomparable to P_2 ($P_1 || P_2$) if $P_1 \not\leq P_2$ and $P_2 \not\leq P_1$. In Fig. 3.1, each block of M_1 is contained in some block of M_6 and hence, $M_6 < M_1$. Since $M_1 \not\leq M_8$ and $M_8 \not\leq M_1$, $M_1 || M_8$. Consider machines M_1 and M_6. If the same sequence of events are applied to both these machines, then given that the state of M_1 is $\{t^1\}$, the state of M_6 ($< M_1$) is $\{t^0, t^1, t^2\}$.

In the literature on state machine decomposition, the less than or equal to relation among machines is referred to as *homomorphism* among machines. So, if $P_1 < P_2$, then P_1 is homomorphic to P_2. If $P_1 < P_2$ and $P_2 < P_1$, then P_1 and P_2 are *isomorphic* to each other.

Observation 1. Given two machines P_1 and P_2 such that $P_1 \leq P_2$, if both machines act on the exact same event sequence, then given the state of P_2, we can uniquely determine the state of P_1.

The set of all machines lesser than a given machine is called the *closed partition set* of the machine. Fig. 3.1 shows the closed partition set of machine $\top$. In the figure, we shown an arrow from one machine P to another machine Q if $P < Q$ and there exists no machine greater than P that is less than Q. Given any two machines in this set, their join and meet are uniquely defined and belong to the closed partition set. Hence, the set of all closed partitions corresponding to a machine, form a lattice under the $\leq$ relation [39]. This lattice is referred to as the *closed partition lattice*.

Note that, in the literature [55], the top-most machine in the closed partition set is shown as the machine with all the states of $\top$ combined and the bottom-most machine is the original machine, i.e., $\top$. However, in this dissertation, similar to Fig. 3.1, the top-most machine is the original machine and the bottom-most machine is the machine containing states in the same block. This is because, we wish to convey the fact that the top-most machine has the maximum information in terms of states and the bottom-most machine has the least information. We now define the lower cover of a machine.

Definition 7. (Lower Cover) Given a machine A, its lower cover, denoted by $\mathcal{C}$, is a set of machines such that:

1. All machines in $\mathcal{C}$ are lesser than A and,
2. There exists no machine C' greater than any machine in $\mathcal{C}$ that is lesser than A.

In Fig. 3.1, the lower cover of $\top$ consist of machines $\{M_1, M_2, M_7, M_8\}$. In [55], the authors present an algorithm to generate the lower cover of a given machine A with time complexity $O(|\Sigma_A||X_A|^2)$ (best-known in terms of time complexity).

3.2 Basics of Linear Coding

In this section, we explain the basic concepts of linear coding, based on the description in [74, 76, 81]. In this dissertation, we assume that we are given n primary servers, among which we want to correct f crash faults using f additional backups. However, in the standard literature for coding theory, it is assumed that we are given k data words and they are encoded onto n

coding words. In this section, we use the standard notation used in coding theory literature. We start with some fundamental concepts from abstract algebra.

3.2.1 Concepts from Abstract Algebra

A *group*, denoted by $(G, +)$, is a non-empty set G with a binary operation "+" that satisfies the following properties:

- Closure: $\forall a, b \in G : a + b \in G$.
- Associativity: $\forall a, b, c \in G : (a + b) + c = a + (b + c)$.
- Identity: There exists an element, denoted by "0", belonging to G such that $\forall a \in G : a + 0 = 0 + a = a$.
- Inverse: $\forall a \in G : \exists a^{-1} \in G : a + a^{-1} = a^{-1} + a = 0$.

A group is called *commutative* or *Abelian* if $\forall a, b \in G : a + b = b + a$. For example, the set of integers Z, is a commutative group with respect to the addition operation, with the identity element being $0 \in Z$ and for all $a \in Z$, the inverse element is $(-a)$. Clearly, $(a) + (-a)$ is equal to 0, i.e, the identity element.

A *ring*, denoted by $(R, +, \cdot)$ is a non-empty set R with two binary operations "+" and "$\cdot$" that satisfies the following properties:

- $(R, +)$ is a commutative group.
- Associativity of "$\cdot$": $\forall a, b, c \in R : (a \cdot b) \cdot c = a \cdot (b \cdot c)$.

- Distributivity: $\forall a, b, c \in R : a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ and $(b + c) \cdot a = (b \cdot a) + (c \cdot a)$.

A ring is called *commutative* if the "$\cdot$" operation is commutative. The set of integers Z with "+" being the addition operation and "$\cdot$" being the multiplication operation is a commutative ring.

A *field* is a commutative ring $(R, +, \cdot)$ in which the non-zero elements from a group with respect to the "$\cdot$" operation. The identity element of the group $(R - \{0\}, \cdot)$ is called the multiplicative identity of F and is denoted by "1". The identity element of $(R, +)$ is called the additive inverse of F and is denoted by "0".

For example, the set of real numbers with respect to addition (+) and multiplication $(\cdot)$ is a field. However, the set of integers Z with respect to addition and multiplication is *not* a field. This is because, all elements of $Z - \{0\}$ do not have an inverse with respect to multiplication. Hence, the non-zero elements of Z do not form a group under "$\cdot$". A field with a finite set of elements is called a *finite field* or a *Galois field.* A finite field with q elements is denoted by $GF(q)$.

A *vector space* over a finite field F is a group V with a "scalar multiplication" that satisfies the following properties:

- $\forall \lambda, \mu \in F, v \in V : (\lambda + \mu)v = \lambda v + \mu v$.
- $\forall \lambda, \mu \in F, v \in V : \lambda(\mu v) = (\lambda \mu)v$.
- $\forall \lambda \in F, v, w \in V : \lambda(v + w) = \lambda v + \lambda w$.
- $\forall v \in V : 1v = v$, where 1 is the multiplicative identity of F.

A set of vectors $v_1, v_2, \ldots v_n \in V$, is called a *generator set* for V if, for all $v \in V$, there exists $\lambda_1, \lambda_2, \ldots, \lambda_n \in F$ such that $v = \lambda_1 v_1 + \lambda_2 v_2 + \ldots \lambda_n v_n$. In other words, *linear combinations* of the elements of the generator set can generate all the elements of the vector space. For example, $F^n = \{(a_1, a_2, \ldots a_n) : a_1, a_2, \ldots \in F\}$ is a vector space over a field F with a generator set $G = \{(1, 0, \ldots 0), (0, 1, \ldots 0), \ldots (0, 0, \ldots 1)\}$.

A set of vectors $v_1, v_2, \ldots v_n \in V$, is called *linearly independent* if the only solution to $\lambda_1 v_1 + \lambda_2 v_2 + \ldots \lambda_n v_n = 0$, is $\lambda_1 = \lambda_2 = \ldots \lambda_n = 0$, where $\lambda_i \in F$. A set of generators that is linearly independent is called a *basis*. Informally, the basis of a vector space is the tightest possible representation of the vector space. For example, the set of prime numbers can represent the entire set of integers. It can been shown that all bases of a vector space have the same cardinality. This cardinality is referred to as the *dimension* of the vector space. A *vector subspace* or *linear subspace* W is a subset of V such that W is a vector space over F with the same binary operations as defined by V.

Consider the vector space F^n over the finite field F. Given any two vectors $v, w \in F^n$, indexed by $[n]$, the *Hamming distance* between them, denoted by $d(v, w)$, is given by $|\{i : v[i] \neq w[i]\}|$. The *hamming norm* of any vector $v \in F^n$, denoted by $|v|$, is $d(v, 0)$. In the following section, we present a brief overview on the construction of finite fields.

3.2.2 Constructing Finite Fields

Since computers use fixed-size words, the computation and algebra in coding must work for these fixed-size words. Assuming each data word to contain w bits, we need to construct finite fields of size 2^w. For $w = 1$, we can simply use the finite field $GF(2) = \{0, 1\}$, where $0, 1 \in Z$, and addition and multiplication are defined modulo 2. In other words, $(+)$ is defined as the XOR of the elements and $(\cdot)$ is defined as the AND of the elements.

For values of w greater than one, we cannot define the field to be the set of integers $\{0 \ldots 2^{w-1}\}$, with addition and multiplication performed modulo 2^w. This is because, the set thus defined is not closed under multiplication for all values of w. For example, consider the set $S = \{0, 1, 2, 3\}$ for $w = 2$, with addition and multiplication defined modulo 4. Clearly, S is not a field because the element 2 has no multiplicative inverse, i.e, there is no element $a \in S$ such that $2a \equiv 1$ (modulo 4).

In this section, we describe a common way to construct fields of size 2^w, based on operations on polynomials in some variable x with coefficients in $GF(2)$. Later, we describe how these polynomials can be mapped into binary words. These are the steps to construct finite fields of size 2^w, denoted by $GF(2^w)$:

1. Construct a primitive polynomial $q(x)$ of degree w whose coefficients are in $GF(2)$. A polynomial $P(x)$ is said to be *irreducible* or *primitive* if the degree of $P(x)$ is greater than zero and $P(x) = a(x) \cdot b(x) \Rightarrow$ deg $a(x) = 0$ or deg $b(x) = 0$. In other words, these polynomials cannot be factored any further. These polynomials can be found in standard texts for coding theory. Some examples are: $x^2 + x + 1$ for $w = 2$, $x^4 + x + 1$ for $w = 4$ and $x^8 + x^4 + x^3 + x^2 + 1$ for $w = 8$.

2. Construct the set $GF(2^w)$ starting with elements 0, 1 and x, and subsequently enumerate the elements of the set by multiplying the last generated element by x, modulo $q(x)$. The enumeration of the set ends when

the number of elements is 2^w. The set thus generated is a finite field and is also denoted by $GF(2)[x]/q[x]$.

Consider the construction of $GF(4)$. Since $w = 2$, $q(x) = x^2 + x + 1$. The first there elements of the field are 0, 1 and x. To generate the next element, multiply the last element by x. Then take the result of this newly generated element, i.e, $x \cdot x = x^2$, modulo $q(x)$. This is standard polynomial long division and x^2 modulo $(x^2 + x + 1) = x + 1$. Hence the elements of $GF(4)$, denoted $GF(2)[x]/(x^2 + x + 1) = \{0, 1, x, x + 1\}$.

Addition of elements belonging to the field is polynomial addition, while multiplication and division is polynomial multiplication and division performed modulo $q(x)$. For example, the sum of two elements x and $x + 1$ belonging to $GF(4)$ is $(1 + 1) \cdot x + 1 = 0 + 1 = 1$, where $0, 1 \in GF(2)$. The product of these two elements is $(x^2 + x)$ modulo $x^2 + x + 1$ which is equal to one.

Given a polynomial $r(x) \in GF(2^w)$, we can map it to a binary word b with w bits by setting the i^{th} bit of b to the coefficient of x^i in $r(x)$. For example, if the field is $GF(4)$, then the polynomial 1, i.e., x^0, is mapped to $b = 01$ since the coefficient of x^0 is 1, while the coefficient of x^1 is 0. Similarly, $x + 1$ is mapped to 11.

Addition of two binary elements $b_1, b_2 \in GF(2^w)$ is simply $b_1\ XOR\ b_2$, since the coefficients of the polynomials all belong to $GF(2)$. To multiply or divide the binary elements, we need to first convert them to the polynomials in the field, perform the multiplication or division and then convert the result to binary. For efficiency, this operation is implemented using look-up tables between the binary elements and the polynomials of the field. In the following section, we describe linear codes, which are essentially vector spaces defined over finite fields.

3.2.3 Linear Codes

Linear codes are the most commonly studied and implemented class of codes owing to their structure which allows for efficient encoding and decoding. Given data or information, the main purpose of coding is to add additional information or redundancy to the given data to make it resilient to failures. Here, we consider two types of failures: (i) *erasures*, in which the data is lost and (ii) *errors*, in which the data is corrupted. First, we formally define a linear code.

Definition 8. (linear code) An $[n, k, d]$ linear code C is a vector subspace of the vector space F^n, defined over a finite field F, where k is the dimension of C and d is the minimum Hamming distance between any two vectors in C.

A *generator matrix* G, of an $[n, k, d]$ linear code C, is a $k \times n$ matrix whose rows form a basis of C. G is called a *systematic* generator matrix if it is of the form $(I|A)$, where I is the $k \times k$ identity matrix, and A is a $k \times (n-k)$ matrix. Given the data words, we use the generator matrix to encode the data to generate words belonging to the code.

Definition 9. (encoding) Given a vector u belonging to F^k we can generate a vector belonging to C by multiplying u with G. The elements of u are referred to as the information/data words and the elements of uG are referred to as the code words.

Hence, given a k-length data word we add information to convert it to an n-length code word. The *rate* of a code, defined as (k/n) captures the redundancy of the code.

For example, the $[3, 2, 2]$ parity code is a linear code over $GF(2)$. From the definition of parity (say even parity of 1s), we know that the code words corresponding to the data words [00], [01], [10] and [11] are $C = \{[000], [011], [101], [110]\}$. Clearly, a basis for C is $\{[101], [011]\}$, since linear combinations of these two elements can generate all the elements in C. For example, $[101] \cdot 1 + [011] \cdot 1 = [110]$. Hence, $k = 2$. Also, the minimum Hamming distance among any two elements of C is two and hence, $d = 2$. A systematic generator matrix for this code is given by,

$$G = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

Let C be an $[n, k, d]$ linear code over F. A *parity-check matrix* of C, is a matrix H, such that for every $c \in F^n$, $c \in C \Leftrightarrow Hc^T = 0$. It can be shown that given a generator matrix G of a code, we can generate a parity-check matrix and vice-versa. In the special case where G is a systematic matrix $(I|A)$, the matrix $H = (-A^T|I)$ can be taken as a parity-check matrix.

Given a data word $u \in F^k$, we first encode u to generate a word $c \in C$. Let us assume that c gets corrupted by some additive noise, i.e., the corrupted word can be represented as $y = c + e$, where $y, c, e \in F^n$. To retrieve the data word u, we need to decode y.

Definition 10. (nearest code word decoding) Consider an $[n, k, d]$ linear code C. Given a vector $y \in F^n$, the goal of a decoder is to find a codeword $c \in C$ that minimizes $d(y, c)$. Equivalently, the decoder needs to find a word $e \in F^n$ with minimum Hamming norm such that $(y - e) \in C$.

It can be shown that, for any $[n, k, d]$ code C over a finite field F, $\forall x \in F^n$, there is at-most one $v \in C$ with $d(c, x)$ less than $\lfloor d/2 \rfloor$. Hence, using nearest code word decoding, we can correct less than or equal to $\lfloor (d-1)/2 \rfloor$ errors in the corrupted word y and generate the correct code word c. Subsequently, we can generate the data word u. We generalize this result in the following theorem.

Theorem 1. An $[n, k, d]$ linear code can *correct* less than or equal to $\lfloor (d-1)/2 \rfloor$ errors, *detect* less than or equal $(d-1)$ errors or *correct* less than or equal to $(d-1)$ erasures.

Definition 11. (MDS Codes) An $[n, k, d]$ linear code is said to be maximum distance separable if $d = n - k + 1$.

Clearly, to correct $d-1$ erasures, we need at least $d-1$ additional units of redundancy. Since $n - k = d - 1$ for MDS codes, they are optimal in terms of the amount of redundancy needed to correct errors or erasures. The $[3, 2, 2]$ parity code is a simple example of an MDS code.

In the following paragraph we describe a standard technique to implement nearest code word decoding, referred to as *syndrome decoding*. Given a code with parity-check matrix H, the syndrome of any word $y \in F^n$ is defined by $S = Hy^T$. Syndrome decoding can be performed through the following steps:

1. Given the word $y \in F^n$, compute its syndrome $S = Hy^T$.

2. Find a word e with minimum Hamming norm such that, $S = He^T$.

From the definition of the parity-check matrix, $Hy^T = He^T \Rightarrow (y-e) \in C$. Since e has the minimum Hamming norm, $(y-e) \in C$ is the code word nearest to the encoded data word. The first step in syndrome decoding involves simple matrix multiplication. The second step, however, is computationally expensive. Many commonly used codes have parity-check matrices with special structures that allow the second step to be performed efficiently. In the following section, we focus on the widely used Reed-Solomon codes.

3.2.3.1 Reed-Solomon Codes

The Reed-Solomon (RS) erasure codes are the only known MDS codes for all possible values of n and k. In this section, we focus on the construction of RS codes and the technique for correcting erasures. There is extensive literature on decoding RS codes to correct *errors* as well [9, 32, 36, 88]. However, due to their high decoding time complexities, we do not use the error-correction routines for RS decoding. In this section, we first describe properties common to all variants of RS coding.

The $k \times n$ generator matrix for RS coding, referred to as the information dispersal matrix B, is chosen to satisfy the following properties:

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

Let D be the data vector and P the encoded vector obtained after multiplying D with B. In the case of erasures, we can first recover the data words using the encoded vector P and the information dispersal matrix B. Erasures are reflected by deleting the corresponding columns from B and P to obtain B' and P' that satisfy the equation, $D \times B' = P'$. For example, if $D = \{d_1, d_2, d_3\}$ and the erasure is in d_2, then the second columns of B and P are deleted to obtain B' and P'. When exactly $n - k$ erasures occur, B' is a $k \times k$ matrix. As mentioned above, any sub-matrix generated by deleting $n-k$ rows from B is an invertible matrix. Hence, matrix B' is guaranteed to be invertible. The data words can be generated as: $P' \times (B')^{-1} = D$. Given the data words, we can regenerate any of the erased code words. If the actual number of erasures are t, then the cost of decoding is $O(kt^2)$ [74].

In this section, we discuss two variants of RS coding. First, we describe the classical Vandermonde RS codes followed by Cauchy RS codes which are the best known implementation of RS codes [11, 76]. The generator matrix for the Vandermonde RS code, is based on the $k \times n$ Vandermonde matrix, in which the $(i, j)^{th}$ element is j^i. For example, for $k = 3, n = 4$, the Vandermonde matrix is,

$$V = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 2^2 & 3^2 \end{bmatrix}$$

The generator matrix for the Vanderomonde RS codes is constructed by performing a sequence of elementary matrix operations to reduce V to the systematic form $(I|S)$, where I is the $k \times k$ identity matrix and S is a $k \times n-k$ matrix. All operations are performed over the finite field $GF(2^w)$, where $n \leq 2^w$.

Assume a set of data blocks that we wish to encode, each containing B bytes of data. Since the data words for Vandermonde RS coding must contain w bits, each data block is divided into $8B/w$ data words of w bits each.

We now study a technique for RS coding, referred to as Cauchy RS coding that is much more efficient. In Cauchy RS coding, the generator or information dispersal matrix is derived from a Cauchy matrix. A $k \times n-k$

Cauchy matrix is defined over the finite field $GF(2^w)$, where $n \leq 2^w$ as follows [76]: Let $X = \{x_1 \ldots x_{n-k}\}$ and $Y = \{y_1 \ldots y_k\}$ be defined such that x_i, y_i are distinct elements of $GF(2^w)$ and $X \cap Y = \phi$. The Cauchy matrix T defined by X and Y has $1/(x_j + y_i)$ as the $(i,j)^{th}$ element. The generator matrix is given by $(I|T)$, where I is the $k \times k$ identity matrix.

In Cauchy RS coding, each element of $GF(2^w)$ can be projected onto a $1 \times w$ vector of bits (refer to [11] for details) such that all operations over $GF(2^w)$ can be replaced with operations over $GF(2)$. The information dispersal matrix is now converted into a $wk \times wn$ matrix. Given a set of data blocks, each containing B bytes, the data block is divided into w *packets* where each packet contains $8B/w$ bits.

These are the main advantages of Cauchy RS coding over Vandermonde RS coding:

1. In Vandermonde RS coding, all operations are performed over the elements of $GF(2^w)$. As seen in section 3.2.2, multiplication and division of elements in the field involve multiple table look-ups. In Cauchy RS coding, encoding simply involves the XOR operation for addition and the AND operation for multiplication. Hence, it is much more efficient.

2. In Vandermonde RS coding, since the data block is divided into words of w bits, the typical values of w are chosen to be 4, 8 and 16 so that 32 and 64 bit machine words can be evenly divided into words. In Cauchy RS coding, since the data block is divided into w packets each containing $8B/w$ bits, w can chosen to be any number as long as $8B/w$ divides 32 or 64.

Chapter 4

Fused State Machines

In this chapter, we present the theory and algorithms of our fusion-based solution for fault tolerance in deterministic finite state machines (referred to as DFSMs or machines). We build a framework for fault tolerance in DFSMs and introduce the concept of an (f, m)-fusion, which is a set of m backup machines that can correct f crash faults or $\lfloor f/2 \rfloor$ Byzantine faults among a given set of machines. Further, we present an algorithm to generate an (f, f)-fusion for a given set of n machines. We ensure that our backups are efficient in terms of the size of their state and event sets. Note that, our solution only requires f backups as compared to the nf backups required by replication. Finally, we present algorithms for the detection and correction of faults that are almost as efficient as replication.

4.1 Framework for Fault Tolerance in DFSMs

In this section, we describe our framework for fault tolerance in DFSMs, using which we can specify the exact number of crash or Byzantine faults that any set of machines can correct. Further, we introduce the concept of an (f, m)-fusion for a set of primaries that is a set of machines that can correct f crash faults, detect f Byzantine faults and correct $\lfloor f/2 \rfloor$ Byzantine faults. Table 4.1 summarizes the notation used in this chapter.

Table 4.1: Symbols/Notation used in this chapter

$\mathcal{P}$	Set of primaries	n	Number of primaries
RCP	Reachable Cross Product	N	Number of states in the RCP
f	No. of crash faults	s	Maximum number of states among primaries
$\mathcal{F}$	Set of fusions/backups	ρ	Average State Reduction in fusion
Σ	Union of primary event-sets	β	Average Event Reduction in fusion

4.1.1 DFSMs and their Reachable Cross Product

A DFSM, denoted by A, consists of a set of states X_A, set of events Σ_A, transition function $\alpha_A : X_A \times \Sigma_A \rightarrow X_A$ and initial state a^0. The size of A, denoted by $|A|$ is the number of states in X_A. A state, $s \in X_A$, is *reachable* iff there exists a sequence of events, which, when applied on the initial state a^0, takes the machine to state s. Consider any two machines, A (X_A, Σ_A, α_A, a^0) and B (X_B, Σ_B, α_B, b^0). Now construct another machine that consists of all the states in the product set of X_A and X_B with the transition function $\alpha'(\{a, b\}, \sigma) = \{\alpha_A(a, \sigma), \alpha_B(b, \sigma)\}$ for all $\{a, b\} \in X_A \times X_B$ and $\sigma \in \Sigma_A \cup \Sigma_B$. This machine ($X_A \times X_B$, $\Sigma_A \cup \Sigma_B$, α', $\{a^0, b^0\}$) may have states that are not reachable from the initial state $\{a^0, b^0\}$. If all such unreachable states are pruned, we get the *reachable cross product* of A and B.

In Fig. 4.1, R is the reachable cross product of A, B and C. Throughout the dissertation, when we just say RCP, we refer to the reachable cross product of the set of primary machines. Given a set of primaries, the number of states in its RCP is denoted by N and its event set, which is the union of the event sets of the primaries is denoted by Σ. Given the state of the RCP, we can determine the state of each of the primary machines and vice versa. However, the RCP has states exponential in n and an event set that is the union of all primary event sets. *Can we generate backup machines that contain fewer*

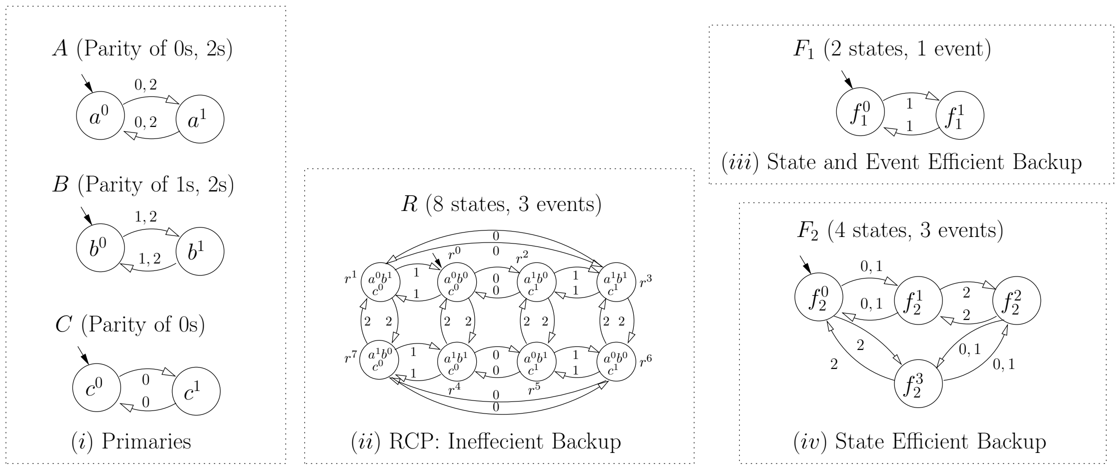


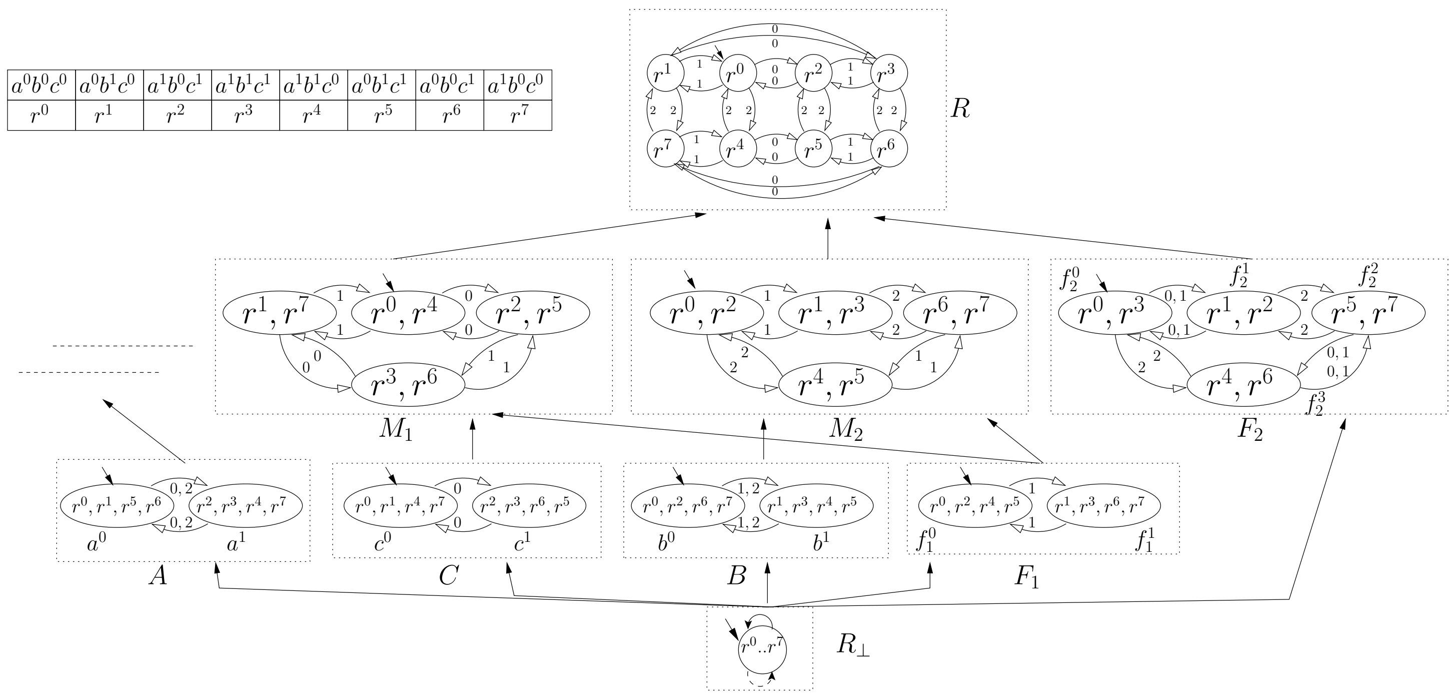
Figure 4.1: Correcting one crash fault among $\{A, B, C\}$ using just one backup.

states and events than the RCP? In the following section, we describe the closed partition set of the RCP.

4.1.2 Decomposition of the Reachable Cross Product

In section 3.1, we describe the decomposition of any give machine A. Given any machine A, we can partition its state space such that the transition function α_A, maps each block of the partition to another block for all events in Σ_A [39, 55]. The set of all the machines generated by partitioning the state space of A is referred to as the closed partition set of A. In this section, we discuss the closed partitions corresponding to the RCP of the primaries. In Fig. 4.2, we show the closed partition set of the RCP of $\{A, B, C\}$ (labeled R).

Consider machine M_2 in Fig. 4.2, generated by combining the states r^0 and r^2 of R. Note that, on event 1, r^0 transitions to r^1 and r^2 transitions to r^3. Hence, we need to combine the states r^1 and r^3. Continuing this procedure,



$a^0b^0c^0$	$a^0b^1c^0$	$a^1b^0c^1$	$a^1b^1c^1$	$a^1b^1c^0$	$a^0b^1c^1$	$a^0b^0c^1$	$a^1b^0c^0$
r^0	r^1	r^2	r^3	r^4	r^5	r^6	r^7

Figure 4.2: Set of machines less than R (all machines not shown due to space constraints).

we obtain the combined states in M_2. Hence, we have *reduced* the RCP to generate M. By combining different pairs of states and by further reducing the machines thus formed, we can construct the entire closed partition set of R.

We can define an order ($\leq$) among any two machines P and Q in this set as follows: $P \leq Q$, if each block of Q is contained in a block of P (shown by an arrow from P to Q). Intuitively, given the state of Q we can determine the state of P. Machines P and Q are incomparable, i.e., $P||Q$, if $P \not\leq Q$ and $Q \not\leq P$. In Fig. 4.2, $F_1 < M_2$, while $M_1||M_2$. Given the state of the primaries, we can determine the state of the RCP and vice versa. Hence, the primary machines are always part of the closed partition set of the RCP (see A, B and

C in Fig. 4.2).

Among the machines shown in Fig. 4.2, some of them, like F_2 (4 states, 3 events) have reduced states, while some like M_1 (4 states, 2 events) and F_1 (2 states, 1 event) have both reduced states and events as compared to R (8 states, 3 events). *Which among these machines can act as backups?* In the following section, we describe the concept of fault graphs and their Hamming distances to answer this question.

4.1.3 Fault Graphs and Hamming Distances

We begin with the idea of a *fault graph* of a set of machines $\mathcal{M}$, for a machine T, where all machines in $\mathcal{M}$ are less than or equal to T. This is a weighted graph and is denoted by $G(T, \mathcal{M})$. The fault graph is an indicator of the capability of the set of machines in $\mathcal{M}$ to correctly identify the current state of T. As described in the previous section, since all the machines in $\mathcal{M}$ are less than or equal to T, the set of states of any machine in $\mathcal{M}$ corresponds to a closed partition of the set of states of T. Hence, given the state of T, we can determine the state of all the machines in $\mathcal{M}$ and vice versa.

Definition 12. (Fault Graph) Given a set of machines $\mathcal{M}$ and a machine $T = (X_T, \Sigma_T, \alpha_T, t^0)$ such that $\forall M \in \mathcal{M} : M \leq T$, the fault graph $G(T, \mathcal{M})$ is a *fully connected weighted graph* where,

- Every node of the graph corresponds to a state in X_T
- The weight of the edge (t^i, t^j) between two nodes, where $t^i, t^j \in X_T$, is the number of machines in $\mathcal{M}$ that have states t^i and t^j in distinct blocks

We construct the fault graph $G(R, \{A\})$, referring to Fig. 4.2. A has two states, $a^0 = \{r^0, r^1, r^5, r^6\}$ and $a^1 = \{r^2, r^3, r^4, r^7\}$. Given just the current

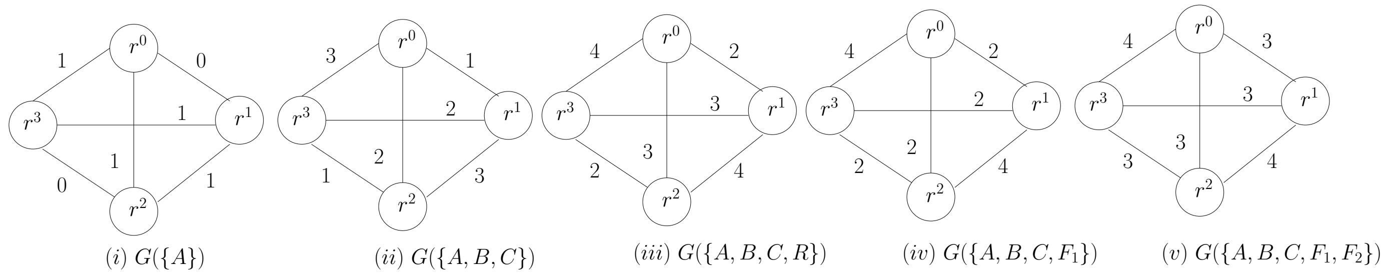

(i) $G(\{A\})$
(ii) $G(\{A, B, C\})$
(iii) $G(\{A, B, C, R\})$
(iv) $G(\{A, B, C, F_1\})$
(v) $G(\{A, B, C, F_1, F_2\})$
r^0
r^1
r^2
r^3

Figure 4.3: Fault Graphs, $G(R, \mathcal{M})$, for sets of machines shown in Fig. 4.2. All eight nodes r^0-r^7 with their edges have not been shown due to space constraints.

state of A, it is possible to determine if R is in state r^0 or r^2 (exact) or one of r^0 and r^1 (ambiguity). Here, A distinguishes between the (r^0, r^2) but not between (r^0, r^1). Hence, in the fault graph $G(R, \{A\})$ in Fig. 4.3 (i), the edge (r^0, r^2) has weight one, while (r^0, r^1) has weight zero. A machine $M \in \mathcal{M}$, is said to *cover* an edge (t^i, t^j) or *separate* the pair of states t^i, t^j, if t^i and t^j lie in separate blocks of M. In Fig. 4.2, A covers (r^0, r^2). In Fig. 4.4 and 4.5, we show an example of the closed partition set and fault graphs for a different set of primaries.

Given the states of $|\mathcal{M}| - x$ machines in $|\mathcal{M}|$, it is always possible to determine if T is in state t^i or t^j iff the weight of the edge (t^i, t^j) is greater than x. Consider the graph shown in Fig. 4.3 (ii). Given the state of any two machines in $\{A, B, C\}$, we can determine if R is in state r^0 or r^2, since the weight of that edge is greater than one, but cannot do the same for the edge (r^0, r^1), since the weight of the edge is one. In coding theory [9, 73], the concept of Hamming distance [38] is widely used to specify the fault tolerance of an erasure code. If an erasure code has minimum Hamming distance greater than d, then it can correct d erasures or $\lfloor d/2 \rfloor$ errors. To understand the fault tolerance of a set of machines, we define a similar notion of distances for the

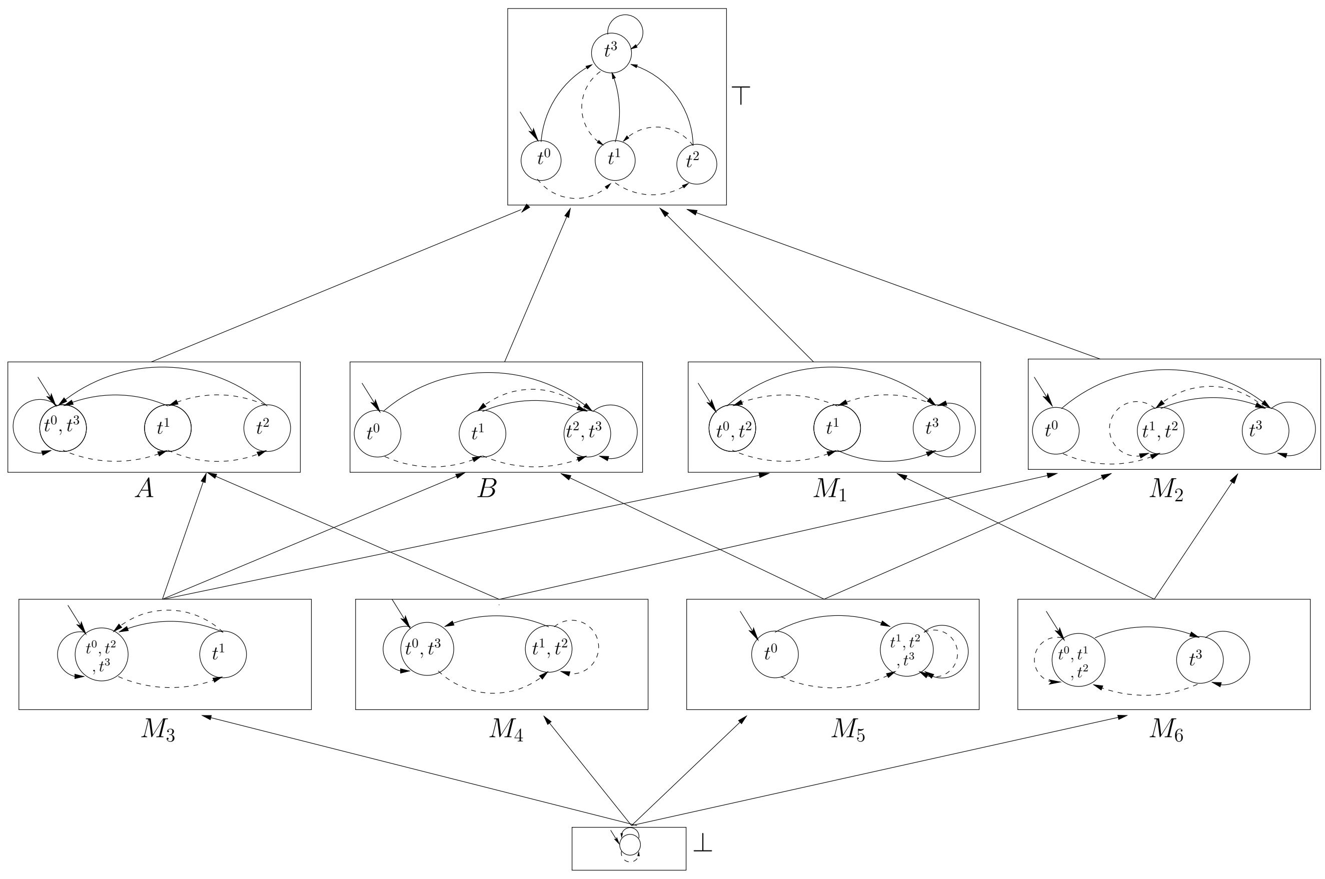


Figure 4.4: Closed partition set for the RCP of $\{A, B\}$.

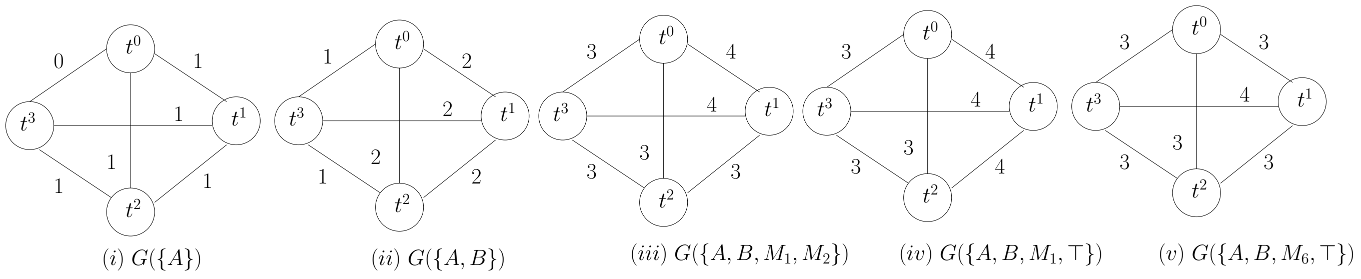

(i) $G(\{A\})$
(ii) $G(\{A, B\})$
(iii) $G(\{A, B, M_1, M_2\})$
(iv) $G(\{A, B, M_1, \top\})$
(v) $G(\{A, B, M_6, \top\})$

Figure 4.5: Fault Graphs for sets of machines shown in Fig. 4.4.

fault graph.

Definition 13. (distance) Given a set of machines $\mathcal{M}$ and their reachable cross product T $(X_T, \Sigma_T, \alpha_T, t^0)$, the distance between any two states $t_i, t_j \in X_T$, denoted by $d(t_i, t_j)$, is the weight of the edge (t_i, t_j) in the fault graph $G(T, \mathcal{M})$. The least distance in $G(T, \mathcal{M})$ is denoted by $d_{min}(T, \mathcal{M})$.

Given a fault graph, $G(T, \mathcal{M})$, the smallest distance between the nodes in the fault graph specifies the fault tolerance of $\mathcal{M}$. Consider the graph, $G(R, \{A, B, C, F_1, F_2\})$, shown in Fig. 4.3 (v). Since the smallest distance in the graph is three, we can remove any two machines from $\{A, B, C, F_1, F_2\}$ and still regenerate the current state of R. As seen before, given the state of R, we can determine the state of any machine less than R. Therefore, the set of machines $\{A, B, C, F_1, F_2\}$ can correct two crash faults.

Theorem 2. A set of machines $\mathcal{M}$, can correct up to f crash faults iff $d_{min}(T, \mathcal{M}) > f$, where T is the reachable cross-product of all machines in $\mathcal{M}$.

Proof. ($\Rightarrow$) Given that $d_{min}(T, \mathcal{M}) > f$, we show that any $\mathcal{M} - f$ machines from $\mathcal{M}$ can accurately determine the current state of T, thereby recovering the state of the crashed machines. Since $d_{min}(T, \mathcal{M}) > f$ at least $f + 1$ machines separate any two states of X_T. Hence, for any pair of states $(t_i, t_j) \in X_T$, even after f crash failures in $\mathcal{M}$, at least one machine remains that can distinguish between t_i and t_j. This implies that it is possible to accurately determine the current state of T by using any $\mathcal{M} - f$ machines from $\mathcal{M}$.

($\Leftarrow$) Given that $d_{min}(T, \mathcal{M}) \le f$, we show that the system cannot correct f crash faults. The condition $d_{min}(T, \mathcal{M}) \le f$ implies that there exists states t_i and t_j in $G(T, \mathcal{M})$ separated by distance k, where $k \le f$. Hence there exist exactly k machines in $\mathcal{M}$ that can distinguish between states $t_i, t_j \in X_T$. Assume that all these k machines crash (since $k \le f$) when T is in either t_i or t_j. Using the states of the remaining machines in $\mathcal{M}$, it is not possible to determine whether T was in state t_i or t_j. Therefore, it is not possible to exactly regenerate the state of any machine in $\mathcal{M}$ using the remaining machines.

$\square$

Byzantine faults may include machines that lie about their state. Consider the machines $\{A, B, C, F_1, F_2\}$ shown in Fig. 4.2. From Fig. 4.3 (v), Let the execution states of the machines A, B, C, F_1 and F_2 be $a^0 = \{r^0, r^1, r^5, r^6\}$, $b^1 = \{r^1, r^3, r^4, r^5\}$, $c^0 = \{r^0, r^1, r^4, r^7\}$, $f_1^0 = \{r^0, r^2, r^4, r^5\}$ and $f_2^0 = \{r^0, r^3\}$ respectively. Since r^0 appears four times (greater than majority) among these states, even if there is one liar we can determine that R is in state r^0. But if R is in state r^0, then B must have been in state b^0 which contains r^0. So clearly, B is lying and its correct state is b^1. Here, we can determine the correct state of the liar, since $d_{min}(R, \{A, B, C, F_1, F_2\}) = 3$, and the majority of machines distinguish between all pairs of states.

Theorem 3. A set of machines $\mathcal{M}$, can correct up to f Byzantine faults iff $d_{min}(T, \mathcal{M}) > 2f$, where T is the reachable cross-product of all machines in $\mathcal{M}$.

Proof. ($\Rightarrow$) Given that $d_{min}(T, \mathcal{M}) > 2f$, we show that any $\mathcal{M} - f$ correct machines from $\mathcal{M}$ can accurately determine the current state of T in spite of f liars. Since $d_{min}(T, \mathcal{M}) > 2f$, at least $2f + 1$ machines separate any two states of X_T. Hence, for any pair of states $t_i, t_j \in X_T$, after f Byzantine failures in $\mathcal{M}$, there will always be at least $f + 1$ correct machines that can distinguish between t_i and t_j. This implies that it is possible to accurately determine the current state of T by simply taking a majority vote.

($\Leftarrow$) Given that $d_{min}(T, \mathcal{M}) \leq 2f$, we show that the system cannot correct f Byzantine faults. $d_{min}(T, \mathcal{M}) \leq 2f$ implies that there exists states

$t_i, t_j \in X_T$ separated by distance k, where $k \leq 2f$. If f among these k machines lie about their state, we have only $k - f$ correct machines remaining. Since, $k - f \leq f$, it is impossible to distinguish the liars from the truthful machines and regenerate the correct state of T. $\square$

In this dissertation, we are concerned only with the fault graph of machines w.r.t the RCP of the primaries $\mathcal{P}$. For notational convenience, we use $G(\mathcal{M})$ instead of $G(RCP, \mathcal{M})$ and $d_{min}(\mathcal{M})$ instead of $d_{min}(RCP, \mathcal{M})$. From theorems 2 and 3, it is clear that a set of n machines $\mathcal{P}$, can correct $(d_{min}(\mathcal{P})-1)$ crash faults and $\lfloor (d_{min}(\mathcal{P}) - 1)/2 \rfloor$ Byzantine faults. Henceforth, we only consider backup machines less than or equal to the RCP of the primaries. In the following section, we describe the theory of such backup machines.

4.1.4 Theory of (f, m)-fusion

To correct faults in a given set of machines, we need to add backup machines so that the fault tolerance of the system (original set of machines along with the backups) increases to the desired value. To simplify the discussion, in the remainder of this dissertation, unless specified otherwise, we mean crash faults when we simply say faults. Given a set of n machines $\mathcal{P}$, we add m backup machines $\mathcal{F}$, each less than or equal to the RCP, such that the set of machines in $\mathcal{A} \cup \mathcal{F}$ can correct f faults. We call the set of m machines in $\mathcal{F}$, an (f, m)-fusion of $\mathcal{P}$. From theorem 2, we know that, $d_{min}(\mathcal{A} \cup \mathcal{F}) > f$.

Definition 14. (Fusion) Given a set of n machines $\mathcal{P}$, we call the set of m machines $\mathcal{F}$, an *(f, m)-fusion* of $\mathcal{P}$, if $d_{min}(\mathcal{A} \cup \mathcal{F}) > f$.

Any machine belonging to $\mathcal{F}$ is referred to as a *fused backup* or just a *fusion*. Consider the set of machines, $\mathcal{P} = \{A, B, C\}$, shown in Fig. 4.1.

From Fig. 4.3 (*ii*), $d_{min}(\{A, B, C\}) = 1$. Hence the set of machines $\mathcal{P}$, cannot correct a single fault. To generate a set of machines $\mathcal{F}$, such that, $\mathcal{P} \cup \mathcal{F}$ can correct two faults, consider Fig. 4.3 (*v*). Since $d_{min}(\{A, B, C, F_1, F_2\}) = 3$, $\{A, B, C, F_1, F_2\}$ can correct two faults. Hence, $\{F_1, F_2\}$ is a (2, 2)-fusion of $\{A, B, C\}$. Note that the set of machines in $\{A, A, B, B, C, C\}$, i.e., replication, is a (2, 6)-fusion of $\{A, B, C\}$.

Any machine in the set $\{A, B, C, F_1, F_2\}$ can at most contribute a value of one to the weight of any edge in the graph $G(\{A, B, C, F_1, F_2\})$. Hence, even if we remove one of the machines, say F_2, from this set, $d_{min}(\{A, B, C, F_1\})$ is greater than one. So $\{F_1\}$ is an (1, 1)-fusion of $\{A, B, C\}$.

Theorem 4. (Subset of a Fusion) Given a set of n machines $\mathcal{P}$, and an (f, m)-fusion $\mathcal{F}$, corresponding to it, any subset $\mathcal{F}' \subseteq \mathcal{F}$ such that $|\mathcal{F}'| = m - t$ is a ($f - t$, $m - t$)-fusion when $t \leq min(f, m)$.

Proof. Since, $\mathcal{F}$ is an (f, m)-fusion of $\mathcal{P}$, $d_{min}(\mathcal{A} \cup \mathcal{F}) > f$. Any machine, $F \in \mathcal{F}$, can at most contribute a value of one to the weight of any edge of the graph, $G(\mathcal{A} \cup \mathcal{F})$. Therefore, even if we remove t machines from the set of machines in $\mathcal{F}$, $d_{min}(\mathcal{A} \cup \mathcal{F}) > f - t$. Hence, for any subset $\mathcal{F}' \subseteq \mathcal{F}$, of size $m - t$, $d_{min}(\mathcal{A} \cup \mathcal{F}') > f - t$. This implies that $\mathcal{F}'$ is an ($f - t$, $m - t$)-fusion of $\mathcal{P}$. □

It is important to note that the converse of this theorem is not true. In Fig. 4.2, while $\{M_2\}$ and $\{F_1\}$ are (1, 1)-fusions of $\{A, B, C\}$, since $d_{min}(\{A, B, C, M_2, F_1\}) = 2$, $\{M_2, F_1\}$ is not a (2, 2)-fusion of $\{A, B, C\}$. We now consider the existence of an (f, m)-fusion for a given set of machines $\mathcal{P}$. Consider the existence of a (2, 1)-fusion for $\{A, B, C\}$ in Fig. 4.2. From Fig. 4.3 (*ii*), $d_{min}(\{A, B, C\}) = 1$. Clearly, R covers each pair of edges in the fault graph. Even if we add R to this set, from Fig. 4.3 (*iii*), $d_{min}(\{A, B, C, R\}) < 3$. Hence, there cannot exist a (2, 1)-fusion for $\{A, B, C\}$.

Theorem 5. (Existence of Fusions) Given a set of n machines $\mathcal{P}$, there exists an $(f,\ m)$-fusion of $\mathcal{P}$ iff $m + d_{min}(\mathcal{P}) > f$.

Proof. ($\Rightarrow$) Assume that there exists an $(f,\ m)$-fusion $\mathcal{F}$ for the given set of machines $\mathcal{P}$. Since, $\mathcal{F}$ is an $(f,\ m)$-fusion of $\mathcal{P}$, $d_{min}(\mathcal{P} \cup \mathcal{F}) > f$. The m machines in $\mathcal{F}$, can at most contribute a value of m to the weight of each edge in $G(\mathcal{P} \cup \mathcal{F})$. Hence, $m + d_{min}(\mathcal{P})$ has to be greater than f.

($\Leftarrow$) Assume that $m + d_{min}(\mathcal{P}) > f$. Consider a set of m machines $\mathcal{F}$, containing m copies of the RCP. These copies contribute exactly m to the weight of each edge in $G(\mathcal{P} \cup \mathcal{F})$. Since, $d_{min}(\mathcal{P}) > f - m$, $d_{min}(\mathcal{P} \cup \mathcal{F}) > f$. Hence, $\mathcal{F}$ is an $(f,\ m)$-fusion of $\mathcal{P}$.

□

Given a set of machines, we now define an order among $(f,\ m)$-fusions corresponding to them.

Definition 15. (Order among $(f,\ m)$-fusions) Given a set of n machines $\mathcal{P}$, an $(f,\ m)$-fusion $\mathcal{F} = \{F_1, ..F_m\}$, is less than another $(f,\ m)$-fusion $\mathcal{G}$, i.e, $\mathcal{F} < \mathcal{G}$, iff the machines in $\mathcal{G}$ can be ordered as $\{G_1, G_2, ..G_m\}$ such that $\forall 1 \leq i \leq m : (F_i \leq G_i) \wedge (\exists j : F_j < G_j)$.

An $(f,\ m)$-fusion $\mathcal{F}$ is *minimal*, if there exists no $(f,\ m)$-fusion $\mathcal{F}'$, such that, $\mathcal{F}' < \mathcal{F}$. It can be seen that, $d_{min}(\{A, B, C, M_2, F_2\}) = 3$, and hence, $\mathcal{F}' = \{M_2, F_2\}$ is a (2, 2)-fusion of $\{A, B, C\}$. We have seen that $\mathcal{F} = \{F_1, F_2\}$, is a (2, 2)-fusion of $\{A, B, C\}$. From Fig. 4.2, since $F_1 < M_2$, $\mathcal{F} < \mathcal{F}'$. In Fig.

4.2, since $R_\perp$ cannot be a fusion for $\{A, B, C\}$, there exists no (2, 2)-fusion less than $\{F_1, F_2\}$. Hence, $\{F_1, F_2\}$ is a minimal (2, 2)-fusion of $\{A, B, C\}$.

We now prove a property of the fusion machines that is crucial for practical applications. Consider a set of primaries $\mathcal{P}$ and an $(f,\ m)$-fusion $\mathcal{F}$ corresponding to it. The client sends updates addressed to the primaries to all the backups as well. We show that events or inputs that belong to distinct set of primaries, can be received in any order at each of the fused backups. This eliminates the need for synchrony at the backups.

Consider a fusion $F \in \mathcal{F}$. Since the states of F are essentially partitions of the state set of the RCP, the state transitions of F are defined by the state transitions of the RCP. For example, machine M_1 in Fig. 4.2 transitions from $\{r^0, r^2\}$ to $\{r^1, r^3\}$ on event 1, because r^0 and r^2 transition to r^1 and r^3 respectively on event 1. Hence, if we show that the state of the RCP is independent of the order in which it receives events addressed to different primaries, then the same applies to the fusions.

Theorem 4.1.1 (Commutativity)**.** *The state of a fused backup after acting on a sequence of events, is independent of the order in which the events are received, as long as the events belong to distinct sets of primaries.*

Proof. We first prove the theorem for the RCP, which is also a valid fused backup. Let the set of primaries be $\mathcal{P} = \{P_1 \ldots P_n\}$. Consider an event e_i that belongs to the set of primaries $\mathcal{S}_i \subseteq \mathcal{P}$. If the RCP is in state r, its next state transition on event e_i depends only on the transition functions of the primaries in $\mathcal{S}_i$. Hence, the state of the RCP after acting on two events e_a and e_b is independent of the order in which these events are received by the

RCP, as long as $\mathcal{S}_a \cup \mathcal{S}_b = \phi$. The proof of the theorem follows directly from this. □

So far, we have presented the framework to understand fault tolerance among machines. Given a set of machines, we can determine if they are a valid set of backups by constructing the fault graph of those machines. In the following section, we present a technique to generate such backups automatically.

4.2 Algorithm to Generate Fused Backup Machines

Given a set of n primaries $\mathcal{P}$, we present the *genFusion* algorithm in Fig. 4.6 to generate an $(f,\ f)$-fusion $\mathcal{F}$ of $\mathcal{P}$. The algorithm takes as input two parameters $\triangle s$ and $\triangle e$ and ensures (if possible) that each machine in $\mathcal{F}$ has at most $(N - \triangle s)$ states and at most $(|\Sigma| - \triangle e)$ events, where N is the number of states in the RCP and Σ is the event set of the RCP. Further, we show that $\mathcal{F}$ is a minimal fusion of $\mathcal{P}$. The algorithm has time complexity polynomial in N.

The *genFusion* algorithm executes f iterations and in each iteration adds a machine to $\mathcal{F}$ that increases $d_{min}(\mathcal{P} \cup \mathcal{F})$ (referred to as d_{min}) by one. At the end of f iterations, d_{min} increases to $f+1$ and hence $\mathcal{P} \cup \mathcal{F}$ can correct f faults. The algorithm ensures that the backup selected in each iteration is optimized for states and events. The algorithms for state reduction and event reduction are presented in Fig. 4.7 and 4.8 respectively. In the following paragraphs, we explain the *genFusion* algorithm in detail, followed by an example to illustrate its working.

In each iteration of the *genFusion* algorithm (Outer Loop), we first

```
genFusion
    Input: Primaries $\mathcal{P}$, faults $f$, state-reduction parameter $\triangle s$,
    event-reduction parameter $\triangle e$;
    Output: $(f, f)$-fusion of $\mathcal{P}$;
    $\mathcal{F} \leftarrow \{\}$;
    //Outer Loop
    for $(i = 1$ to $f)$
        Identify weakest edges in fault graph $G(\mathcal{P} \cup \mathcal{F})$;
        $\mathcal{M} \leftarrow \{RCP(\mathcal{P})\}$;
        //State Reduction Loop
        for $(j = 1$ to $\triangle s)$
            $\mathcal{S} \leftarrow \{\}$;
            for $(M \in \mathcal{M})$
                $\mathcal{S} = \mathcal{S} \cup reduceState(M)$;
            $\mathcal{M}$ = All machines in $\mathcal{S}$ that increment $d_{min}(\mathcal{P} \cup \mathcal{F})$;
        //Event Reduction Loop
        for $(j = 1$ to $\triangle e)$
            $\mathcal{E} \leftarrow \{\}$;
            for $(M \in \mathcal{M})$
                $\mathcal{E} = \mathcal{E} \cup reduceEvent(M)$;
            $\mathcal{M}$ = All machines in $\mathcal{E}$ that increment $d_{min}(\mathcal{P} \cup \mathcal{F})$;
        //Minimality Loop
        $M \leftarrow$ Any machine in $\mathcal{M}$;
        while (all states of $M$ have not been combined)
            $\mathcal{C} \leftarrow reduceState(M)$;
            $M$= Any machine in $\mathcal{C}$ that increments $d_{min}(\mathcal{P} \cup \mathcal{F})$;
        $\mathcal{F} \leftarrow \{M\} \bigcup \mathcal{F}$;
    return $\mathcal{F}$;
```

Figure 4.6: Algorithm to generate an (f, f)-fusion for a given set of primaries.

```
reduceState
    Input: Machine $P$ with state set $X_P$, event set $\Sigma_P$
    and transition function $\alpha_P$;
    Output: Largest Machines $< P$ with $\leq |X_P| - 1$ states;
    $\mathcal{B} = \{\}$;
    for $(s_i, s_j \in X_P)$
        //combine states $s_i$ and $s_j$
        Set of states, $X_B = X_P$ with $(s_i, s_j)$ combined;
        $\mathcal{B} = \mathcal{B} \cup \{$Largest machine consistent with $X_B\}$;
    return Incomparable machines in $\mathcal{B}$;
```

Figure 4.7: Algorithm to generate reduced-state machines.

identify the set of weakest edges (lowest weight) in $\mathcal{P} \cup \mathcal{F}$ and then find a machine that covers these edges, thereby increasing d_{min} by one. We start with the RCP, since it always increases d_{min}. The 'State Reduction Loop' and the 'Event Reduction Loop' successively reduce the states and events of the RCP. Finally the 'Minimality Loop' searches as deep into the closed partition set of the RCP as possible for a reduced-state machine, without explicitly constructing the lattice.

State Reduction Loop This loop uses the *reduceState* algorithm in Fig. 4.7 to iteratively generate machines with fewer states than the RCP that increase d_{min} by one. The *reduceState* algorithm, takes as input, a machine P and generates a set of machines in which at least two states of P are combined. For each pair of states s_i, s_j in X_P, the *reduceState* algorithm, first creates a partition of blocks in which (s_i, s_j) are combined and then constructs the largest machine consistent with this partition. Note that, 'largest' is based on the order specified in section 4.1.2. This procedure is repeated for all pairs in X_P and the incomparable machines among them are returned. At the end

```
reduceEvent
    Input: Machine $P$ with state set $X_P$, event set $\Sigma_P$
    and transition function $\alpha_P$;
    Output: Largest Machines $< P$ with $\leq |\Sigma_P| - 1$ events;
    $\mathcal{B} = \{\}$;
    for ($\sigma \in \Sigma_P$)
        Set of states, $X_B = X_P$;
        //combine states to self-loop on $\sigma$
        for ($s \in X_B$)
            $s = s \cup \alpha_P(s, \sigma)$;
        $\mathcal{B} = \mathcal{B} \cup \{$Largest machine consistent with $X_B\}$;
    return Incomparable machines in $\mathcal{B}$;
```

Figure 4.8: Algorithm to generate reduced-event machines.

of $\triangle s$ iterations of the state reduction loop, we generate a set of machines $\mathcal{M}$ each of which increases d_{min} by one and contains at most $(N - \triangle s)$ states, if such machines exist.

Event Reduction Loop Starting with the state reduced machines in $\mathcal{M}$, the event reduction loop uses the *reduceEvent* algorithm in Fig. 4.8 to generate reduced event machines that increase d_{min} by one. The *reduceEvent* algorithm, takes as input, a machine P and generates a set of machines that contain at least one event less than Σ_P. To generate a machine less than any given input machine P, that does not contain an event σ in its event set, the *reduceEvent* algorithm combines the states such that they loop onto themselves on σ. The algorithm then constructs the largest machine that contains these states in the combined form. This machine, in effect, ignores σ. This procedure is repeated for all events in Σ_P and the incomparable machines among them are returned. At the end of $\triangle e$ iterations of the event reduction loop, we generate a set of machines $\mathcal{M}$ each of which increases d_{min} by one and contains at most $(N - \triangle s)$ states and at most $(|\Sigma| - \triangle e)$ events, if such machines exist.

Minimality Loop This loop picks any machine M among the state and event reduced machines in $\mathcal{M}$ and uses the *reduceState* algorithm iteratively to generate a machine less than M that increases d_{min} by one until no further state reduction is possible i.e., all the states of M have been combined. Unlike the state reduction loop (which also uses the *reduceState* algorithm), in the minimality loop we never exhaustively explore all state reduced machines. After each iteration of the minimality loop, we only pick *one* machine that increases d_{min} by one.

Note that, in all three of these inner loops, if in any iteration, no reduction is achieved, then we simply exit the loop with the machines generated in the previous iteration. We use the example in Fig. 4.2 with $\mathcal{P} = \{A, B, C\}, f = 2, \triangle s = 1$ and $\triangle e = 1$, to explain the *genFusion* algorithm. Since $f = 2$, there are two iterations of the outer loop and in each iteration we generate one machine. Consider the first iteration of the outer loop. Initially, $\mathcal{F}$ is empty and we need to add a machine that covers the weakest edges in $G(\{A, B, C\})$.

To identify the weakest edges, we need to identify the mapping between the states of the RCP and the states of the primaries. For example, in Fig. 4.2, we need to map the states of the RCP to A. The starting states are always mapped to each other and hence r^0 is mapped to a^0. Now r^0 on event 0 transitions to r^2, while a^0 on event 0 transitions to a^1. Hence, r^2 is mapped to a^1. Continuing this procedure for all states and events, we obtain the mapping shown, i.e, $a^0 = \{r^0, r^1, r^5, r^6\}$ and $a^1 = \{r^2, r^3, r^4, r^7\}$. Following this procedure for all primaries, we can identify the weakest edges in $G(\{A, B, C\})$ (Fig. 4.3 (*ii*)). In Fig. 4.2, M_1, M_2 and F_2 are some of the largest incomparable machines that contain at least one state less than the RCP (the entire set is too large to be enumerated here). All three of these machines increase d_{min} and at the end of the one and only iteration of the state reduction loop, $\mathcal{M}$ will contain at least these three machines.

The event reduction loop tries to find machines with fewer events than the machines in $\mathcal{M}$. For example, to generate a machine less than M_2 that does not contain, say event 2, the *reduceEvent* algorithm combines the blocks of M_2 such that they do not transition on event 2. Hence, $\{r^0, r^2\}$ in M_2 is combined with $\{r^4, r^5\}$ and $\{r^1, r^3\}$ is combined with $\{r^6, r^7\}$ to generate machine F_1 that does not act on event 2. The only machine less than M_2 that does not act on event 1 is $R_\perp$. Since the *reduceEvent* algorithm returns incomparable machines, only F_1 is returned when M_1 is the input. Similarly, with M_2 as input, the *reduceEvent* algorithm returns $\{C, F_1\}$ and with F_2 as input it returns $R_\perp$. Among these machines only F_1 increases d_{min}. For example, C does not cover the weakest edge (r^0, r^1) of $G(\mathcal{P})$. Hence, at the end of the one and only iteration of the event reduction loop, $\mathcal{M} = \{F_1\}$.

As there exists no machine less than F_1, that increases d_{min}, at the end of the minimality loop, $M = F_1$. Similarly, in the second iteration of the outer loop $M = F_2$ and the *genFusion* algorithm returns $\{F_1, F_2\}$ as the fusion machines that increases d_{min} to three. Hence, using the *genFusion* algorithm, we have automatically generated the backups F_1 and F_2 shown in Fig. 4.1. In the worst case, there may exist no efficient backups and the *genFusion* algorithm may just return a set of f copies of the RCP. However, our results in section 6.3 indicate that for many examples, efficient backups do exist.

4.2.1 Properties of the *genFusion* Algorithm

In this section, we prove properties of the *genFusion* algorithm with respect to: (i) the number of fusion/backup machines (ii) the number of states in each fusion machine, (iii) the number of events in each fusion machine and (iv) the minimality of the set of fusion machines $\mathcal{F}$. We first introduce concepts that are relevant to the proof of these properties.

Lemma 1. Given a set of primary machines $\mathcal{P}$, $d_{min}(\mathcal{P}) = 1$.

Proof. Given the state of all the primary machines, the state of the RCP can be uniquely determined. Hence, there is at least one machine among the primaries that distinguishes between each pair of states in the RCP and so, $d_{min}(\mathcal{P}) \geq 1$. In section 1.3, we state our assumption that the set of machines in $\mathcal{P}$ cannot correct a single fault and this implies that, $d_{min}(\mathcal{P}) \leq 1$. Hence, $d_{min}(\mathcal{P}) = 1$. □

Lemma 2. Given a set of primary machines $\mathcal{P}$, let $\mathcal{F}'$ be an $(f,\ f)$-fusion of $\mathcal{P}$. Each fusion machine $F \in \mathcal{F}'$ has to cover the weakest edges in $G(\mathcal{P})$.

Proof. From lemma 1, the weakest edges of $G(\mathcal{P})$ have weight equal to one. Since $\mathcal{F}'$ is an $(f,\ f)$-fusion of $\mathcal{P}$, $d_{min}(\mathcal{P} \cup \mathcal{F}') > f$. Also, each machine in $\mathcal{F}'$ can increase the weight of any edge by at most one. Hence, all f machines in $\mathcal{F}'$ have to cover the weakest edges in $G(\mathcal{P})$. □

Let the weakest edges of $G(\mathcal{P} \cup \mathcal{F})$ at the start of the i^{th} iteration of the outer loop of the *genFusion* algorithm be denoted E_i. In the following lemma, we show that the set of weakest edges does not decrease with each iteration.

Lemma 3. In the *genFusion* algorithm, for any two iterations i and j, if $i < j$, then $E_i \subseteq E_j$.

Proof. Let the value of d_{min} for the i^{th} iteration be d and the edges with this weight be E_i. Any machine added to $\mathcal{F}$ can at most increase the weight of each edge by one and it has to increase the weight of all edges in E_i by one. So, d_{min} for the $(i+1)^{th}$ iteration is $d+1$ and the weight of the edges in E_i will increase to $d+1$. Hence, E_i will be among the weakest edges in the $(i+1)^{th}$ iteration, or in other words, $E_i \subseteq E_{i+1}$. This trivially extends to the result: for any two iterations numbered i and j of the *genFusion* algorithm, if $i < j$, then $E_i \subseteq E_j$. □

We now prove one of the main theorems of this dissertation.

Theorem 6. (Fusion Algorithm) Given a set of n machines $\mathcal{P}$, the *genFusion* algorithm generates a set of machines $\mathcal{F}$ such that:

1. (Correctness) $\mathcal{F}$ is an (f, f)-fusion of $\mathcal{P}$.

2. (State & Event Efficiency) If each machine in $\mathcal{F}$ has greater than $(N - \triangle s)$ states and $(|\Sigma| - \triangle e)$ events, then no (f, f)-fusion of $\mathcal{P}$ contains a machine with less than or equal to $(N - \triangle s)$ states and $(|\Sigma| - \triangle e)$ events.

3. (Minimality) $\mathcal{F}$ is a minimal (f, f)-fusion of $\mathcal{P}$.

Proof. We prove each of the mentioned properties:

1. From lemma 1, $d_{min}(\mathcal{P}) = 1$. Starting with the *RCP*, which always increases d_{min} by one, we add one machine in each iteration to $\mathcal{F}$ that increases by $d_{min}(\mathcal{P} \cup \mathcal{F})$ by one. Hence, at the end of f iterations of the *genFusion* algorithm, we add exactly f machines to $\mathcal{F}$ that increase d_{min} to $f + 1$. Hence, $\mathcal{F}$ is an (f, f)-fusion of $\mathcal{P}$.

2. Assume that each machine in $\mathcal{F}$ has greater than $(N - \triangle s)$ states and $(|\Sigma| - \triangle e)$ events. Let there be another $(f,\, f)$-fusion of $\mathcal{P}$ that contains a machine F' with less than or equal to $(N - \triangle s)$ states and $(|\Sigma| - \triangle e)$ events. From lemma 2, F' covers the weakest edges in $G(\mathcal{P})$. However, in the first iteration of the outer loop, the *genFusion* algorithm searches exhaustively for a fusion with less than or equal to $(N - \triangle s)$ states and $(|\Sigma| - \triangle e)$ events that covers the weakest edges in $G(\mathcal{P})$. Hence, if such a machine F' existed, then the algorithm would have chosen it.

3. Let there be an $(f,\, f)$-fusion $\mathcal{G} = \{G_1, ..G_f\}$ of $\mathcal{P}$, such that $\mathcal{G}$ is less than $(f,\, f)$-fusion $\mathcal{F} = \{F_2, F_1, ..., F_f\}$. Hence $\forall j : G_j \leq F_j$. Let $G_i < F_i$ and let E_i be the set of edges that needed to be covered by F_i. It follows from the *genFusion* algorithm, that G_i does not cover at least one edge say e in E_i (otherwise the algorithm would have returned G_i instead of F_i). From lemma 3, it follows that if e is covered by k machines in $\mathcal{F}$, then e has to be covered by k machines in $\mathcal{G}$. We know that there is a pair of machines F_i, G_i such that F_i covers e and G_i does not cover e. For all other pairs F_j, G_j if G_j covers e then F_j covers e (since $G_j \leq F_j$). Hence e can be covered by no more than $k - 1$ machines in $\mathcal{G}$. This implies that $\mathcal{G}$ is not an $(f,\, f)$-fusion of $\mathcal{P}$.

$\square$

4.2.2 Time Complexity of the *genFusion* Algorithm

The time complexity of the *genFusion* algorithm is the sum of the time complexities of the inner loops multiplied by the number of iterations, f. We analyze the time complexity of each of the inner loops. Let the set of machines in $\mathcal{M}$ at the start of the i^{th} iteration of the outer loop be denoted $\mathcal{M}_i$.

State Reduction Loop: The time complexity of the state reduction loop for the i^{th} iteration of the outer loop is T_1+T_2, where T_1 is the time complexity to reduce the states of the machines in $\mathcal{M}_i$ and T_2 is the time complexity to find the machines among $\mathcal{S}$ that increment d_{min}. First, let us consider T_1. Note that, initially $\mathcal{M}$, i.e, $\mathcal{M}_1$, contains only the RCP with $O(N)$ states and for any iteration of the state reduction loop, each of the machines in $\mathcal{M}_i$ has $O(N)$ states. Given a machine M with $O(N)$ states, the *reduceState* algorithm generates machines with fewer states than M. For each pair of states in M, the time complexity to generate the largest closed partition that contains these states in a combined block is just $O(N|\Sigma|)$. Since there are $O(N^2)$ pairs of states in M, the time complexity of the *reduceState* algorithm is $O(N^3|\Sigma|)$. Hence, $T_1 = O(|\mathcal{M}_i|N^3|\Sigma|)$.

Now, we consider T_2. Since, there are $O(N^2)$ pairs of states in each machine in $\mathcal{M}_i$, the *reduceState* algorithm returns $O(N^2)$ machines. So, $|\mathcal{S}| = O(N^2|\mathcal{M}_i|)$. Since there are $O(N^2)$ nodes in the fault graph of $G(\mathcal{P} \cup \mathcal{F})$, given any machine in $\mathcal{S}$, the time complexity to check if it increments d_{min} is $O(N^2)$. Hence, $T_2 = O(|\mathcal{S}|N^2) = O(N^4|\mathcal{M}_i|)$. So, the time complexity of each iteration of the state reduction loop is $T_1 + T_2 = O(|\mathcal{M}_i|N^3|\Sigma| + N^4|\mathcal{M}_i|)$.

Since the *reduceState* algorithm generates $O(N^2)$ machines per machine in $\mathcal{M}_i$, $|\mathcal{M}_{i+1}| = N^2|\mathcal{M}_i|$. In the first iteration $\mathcal{M}$ just contains the RCP and $|\mathcal{M}_1| = 1$. Hence, the time complexity of the state reduction loop is, $O((N^3|\Sigma| + N^4)(1 + N^2 + N^4 \ldots + N^{2(\triangle s-1)})) = O((N^3|\Sigma| + N^4)(\frac{N^{2\triangle s}-1}{N^2-1})$ (the series is a geometric progression). This reduces to $O(N^{\triangle s+1}|\Sigma| + N^{\triangle s+2})$. Also, $\mathcal{M}$ contains $O(N^{2\triangle s})$ machines at the end of the state reduction loop.

Event Reduction Loop: The time complexity analysis for the event reduction loop is similar, except for the fact that the *reduceEvent* algorithm iterates through $|\Sigma|$ events of each machine in $\mathcal{M}$ and returns $O(|\Sigma|)$ machines per machine in $\mathcal{M}$. Also, while the state reduction loop starts with just one machine in $\mathcal{M}$, the event reduction loop starts with $O(N^{2\triangle s})$ machines in $\mathcal{M}$. Hence, the time complexity of each iteration of the event reduction loop is $O((N|\Sigma|^2 + N^2|\Sigma|)(N^{2\triangle s})(1 + |\Sigma| + |\Sigma|^2 \ldots + |\Sigma|^{\triangle e-1})) = O((N|\Sigma|^2 + N^2|\Sigma|)(N^{2\triangle s})(\frac{|\Sigma|^{\triangle e}-1}{|\Sigma|-1})) = O(N^{\triangle s+1}|\Sigma|^{\triangle e+1} + N^{\triangle s+2}|\Sigma|^{\triangle e})$.

Minimality Loop: In the minimality loop, we use the *reduceState* algorithm, but only select one machine per iteration. Also, in each iteration of the minimality loop, the number of states in M is at least one less than than the number of states in M for the previous iteration. Hence, the minimality loop executes $O(N)$ iterations with total time complexity, $O((N^3|\Sigma| + N^4)(N)) = O(N^4|\Sigma| + N^5)$.

Since there are f iterations of the outer loop, the time complexity of the *genFusion* algorithm is, $O(fN^{\triangle s+1}|\Sigma| + fN^{\triangle s+2} + fN^{\triangle s+1}|\Sigma|^{\triangle e+1} + fN^{\triangle s+2}|\Sigma|^{\triangle e} + fN^4|\Sigma| + fN^5)$. This reduces to,

$$O(fN^{\triangle s+1}|\Sigma|^{\triangle e+1} + fN^{\triangle s+2}|\Sigma|^{\triangle e} + fN^4|\Sigma| + fN^5)$$

Observation 2. For parameters $\triangle s = 0$ and $\triangle e = 0$, the *genFusion* algorithm generates a minimal (f, f)-fusion of $\mathcal{P}$ with time complexity $O(fN^4|\Sigma|+fN^5)$. Hence, the time complexity is polynomial in the number of states of the RCP.

If there are n primaries each with $O(s)$ states, then N is $O(s^n)$. Hence, the time complexity of the *genFusion* algorithm reduces to $O(s^n|\Sigma|f)$. Even though the time complexity of generating the fusions is exponential in n, note that the fusions have to be generated only once. Further, in the following section, we present an incremental approach for the generation of fusions that improves the time complexity by a factor of $O(\rho^n)$ for constant values of ρ,

```
incFusion
    Input: Primaries $\mathcal{P} = \{P_1, P_2, \dots P_n\}$, faults $f$,
    state-reduction parameter $\triangle s$, event-reduction parameter $\triangle e$;
    Output: $(f,\ f)$-fusion of $\mathcal{P}$;
    $\mathcal{F} \leftarrow \{P_1\}$;
    for $(i = 2$ to $n)$
        $\mathcal{N} \leftarrow \{P_i\} \cup RCP(\mathcal{F})$;
        $\mathcal{F} \leftarrow genFusion(\mathcal{N}, f, \triangle s, \triangle e)$;
    return $\mathcal{F}$;
```

Figure 4.9: Incremental fusion algorithm.

where ρ is the average state reduction achieved by fusion, i.e., (N/Average size of a fusion).

4.2.3 Incremental Approach to Generate Fusions

In Fig. 4.9, we present an incremental approach to generate the fusions, referred to as the *incFusion* algorithm, in which we may never have to reduce the RCP of all the primaries. In each iteration, we generate the fusion corresponding to a new primary and the RCP of the (possibly small) fusions generated for the set of primaries in the previous iteration.

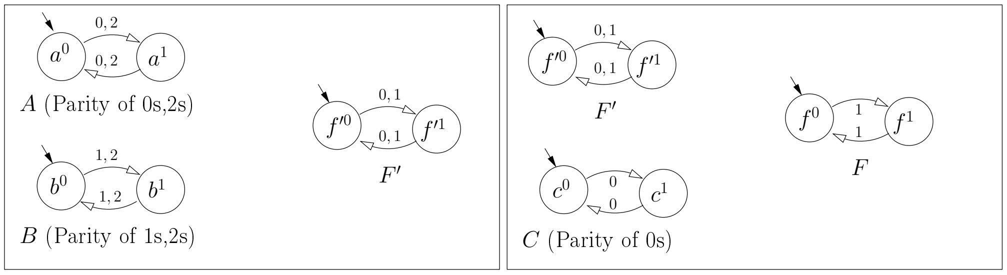


Figure 4.10: Incremental Approach: first generate F' and then F.

In Fig. 4.10, rather than generate a fusion by reducing the 8-state RCP

of $\{A, B, C\}$, we can reduce the 4-state RCP of $\{A, B\}$ to generate fusion F' and then reduce the 4-state RCP of $\{C, F'\}$ to generate fusion F. In the following paragraph, we present the proof of correctness for the incremental approach and show that it has time complexity $O(\rho^n)$ times better than that of the *genFusion* algorithm, where ρ is the average state reduction achieved by fusion.

Theorem 7. Given a set of n machines $\mathcal{P}$, the *incFusion* algorithm generates an $(f,\ f)$-fusion of $\mathcal{P}$.

Proof. We prove the theorem using induction on the variable i in the algorithm. For the base case, i.e., $i = 2$, $\mathcal{N} = \{P_1, P_2\}$ (since $RCP(\{P_1\}) = P_1$). Let the $(f,\ f)$-fusion generated by the *genFusion* algorithm for $\mathcal{N} = \{P_1, P_2\}$ be denoted $\mathcal{F}^1$. For $i = 3$, let the $(f,\ f)$-fusion generated for $\mathcal{N} = \{P_3, RCP(\mathcal{F}^1)\}$ be denoted $\mathcal{F}^2$. We show that $\mathcal{F}^2$ is an $(f,\ f)$-fusion of $\{P_1, P_2, P_3\}$. Assume f crash faults among $\{P_1 P_2, P_3\} \cup \mathcal{F}^2$. Clearly, less than or equal to f machines in $\{P_3\} \cup \mathcal{F}^2$ have crashed. Since $\mathcal{F}^2$ is an $(f,\ f)$-fusion of $\{P_3, RCP(\mathcal{F}^1)\}$, we can generate the state of all the machines in $RCP(\mathcal{F}^1)$ and the state of the crashed machines among $\{P_3\} \cup \mathcal{F}^2$. Similarly, less than or equal to f machines have crashed among $\{P_1, P_2\}$. Hence, using the state of the available machines among $\{P_1, P_2\}$ and the states of all the machines in $\mathcal{F}^1$ we can generate the state of the crashed machines among $\{P_1, P_2\}$.

Induction Hypothesis: Assume that the set of machines $\mathcal{F}^i$, generated in iteration i, is an $(f,\ f)$-fusion of $\{P_1 \dots P_{i+1}\}$. Let the $(f,\ f)$-fusion of $\{P_{i+2}, RCP(\mathcal{F}^i)\}$ generated in iteration $i+1$ be denoted $\mathcal{F}^{i+1}$. To prove: $\mathcal{F}^{i+1}$ is an $(f,\ f)$-fusion of $\{P_1 \dots P_{i+2}\}$. The proof is similar to that for the base case. Using the state of the available machines in $\{P_{i+2}\} \cup \mathcal{F}^{i+1}$, we can generate the state of all the machines in $\mathcal{F}^i$ and $\{P_{i+2}\} \cup \mathcal{F}^{i+1}$. Subsequently, we can generate the state of the crashed machines in $\{P_1 \ldots P_{i+1}\}$. $\Box$

From observation 2, the *genfusion* algorithm has time complexity, $O(fN^4|\Sigma| + fN^5)$ (assuming $\triangle s = 0$ and $\triangle e = 0$ for simplicity). Hence, if the size of $\mathcal{N}$ in the i^{th} iteration of the *incFusion* algorithm is denoted by N_i, then the time complexity of the *incFusion* algorithm, T_{inc} is given by the expression $\Sigma_{i=2}^{i=n} O(fN_i^4|\Sigma| + fN_i^5)$.

Let the number of states in each primary be s. For $i = 2$, the primaries are $\{P_1, P_2\}$ and $N_1 = O(s^2)$. For $i = 3$, the primaries are $\{RCP(\mathcal{F}^1), P_3\}$. Note that $RCP(\mathcal{F}^1)$ is also a fusion machine. Since we assume an average reduction of ρ (size of RCP of primaries/average size of each fusion), the number of states in $RCP(\mathcal{F}^1)$ is $O(s^2/\rho)$. Hence, $N_2 = O(s^3/\rho)$. Similarly, $N_3 = O(s^4/\rho^2)$ and $N_i = O(s^{i+1}/\rho^{i-1})$. Hence, $T_{inc} = O(|\Sigma| f \Sigma_{i=2}^{i=n} s^{4i+4}/\rho^{4i-4} + f\Sigma_{i=2}^{i=n} s^{5i+5}/\rho^{5i-5}) = O(|\Sigma| f s^4 \rho^4 \Sigma_{i=2}^{i=n} (s/\rho)^{4i} + f s^5 \rho^5 \Sigma_{i=2}^{i=n} (s/\rho)^{5i})$. This is the sum of a geometric progression and hence,

$$T_{inc} = O(|\Sigma| f s^4 \rho^4 (s/\rho)^{4n} + f s^5 \rho^5 (s/\rho)^{5n})$$

Assuming ρ and s are constants, $T_{inc} = O(f|\Sigma|s^n/\rho^n + fs^n/\rho^n)$. Note that, the time complexity of the *genFusion* algorithm in Fig. 4.6 is $O(f|\Sigma|s^n + fs^n)$. Hence, the *incFusion* algorithm achieves $O(\rho^n)$ savings in time complexity over the *genFusion* algorithm. In the following section, we consider the event-based decomposition of machines, based on the *reduceEvent* algorithm presented in Fig. 4.8.

4.3 Event-Based Decomposition of Machines

In this section, we explore the problem of replacing a given machine M with two or more machines, each containing fewer events than M. We present an algorithm to generate such event-reduced machines with time complexity polynomial in the size of M. This is important for applications with limits on the number of events each individual process running a DFSM can service. Note that, the contributions in this section are independent of fault tolerance. We first define the notion of event-based decomposition.

Definition 16. A *(k,e)-event decomposition* of a machine $M(X_M, \alpha_M, \Sigma_M, m^0)$ is a set of k machines $\mathcal{E}$, each less than M, such that $d_{min}(M, \mathcal{E}) > 0$ and $\forall P(X_P, \alpha_P, \Sigma_P, p^0) \in \mathcal{E}$, $|\Sigma_P| \leq |\Sigma_M| - e$.

As $d_{min}(M, \mathcal{E}) > 0$, given the state of the machines in $\mathcal{E}$, the state of M can be determined. So, the machines in $\mathcal{E}$, each containing at most $|\Sigma_M| - e$ events, can effectively replace M. In Fig. 4.11, we present the *eventDecompose* algorithm that takes as input, machine M, parameter e, and returns a (k,e)-event decomposition of M (if it exists) for some $k \leq |X_M|^2$.

In each iteration, Loop 1 generates machines that contain at least one event less than the machines of the previous iteration. So, starting with M in the first iteration, at the end of e iterations, $\mathcal{M}$ contains the set of largest machines less than M, each containing at most $|\Sigma_M| - e$ events.

Loop 2, iterates through each machine P generated in the previous iteration, and uses the *reduceEvent* algorithm presented in Fig. 4.8 to generate the set of largest machines less than P containing at least one event less than Σ_P. Loop 3 constructs an event-decomposition $\mathcal{E}$ of M, by iteratively adding at least one machine from $\mathcal{M}$ to separate each pair of states in M, thereby

```
eventDecompose
    Input: Machine $M$ with state set $X_M$, event set $\Sigma_M$
    and transition function $\alpha_M$;
    Output: ($k$,$e$)-event decomposition of $M$ for
    some $k \leq |X_M|^2$;
    $\mathcal{M} = \{M\}$;
    for ($j = 1$ to $e$) //Loop 1
        $\mathcal{G} \leftarrow \{\}$;
        for ($P \in \mathcal{M}$) //Loop 2
            $\mathcal{G} = \mathcal{G} \cup reduceEvent(P)$;
        $\mathcal{M} = \mathcal{G}$;
    $\mathcal{E} \leftarrow \{\}$;
    for ($m_i, m_j \in X_M$) //Loop 3
        if ($\exists E \in \mathcal{M} : E$ separates $m_i, m_j$)
            $\mathcal{E} \leftarrow \mathcal{E} \cup \{E\}$;
        else
            return $\{\}$;
    return $\mathcal{E}$;
```

Figure 4.11: Algorithm for the event-based decomposition of a machine.

ensuring that $d_{min}(\mathcal{E}) > 0$. Since each machine added to $\mathcal{E}$ can separate more than one pair of states, an efficient way to implement Loop 3 is to check for the pairs that still need to be separated in each iteration and add machines until no pair remains.

Let the 4-event machine M shown in Fig. 4.12 be the input to the *eventDecompose* algorithm with $e = 1$. In the first and only iteration of Loop 1, $P = M$ and the *reduceEvent* algorithm generates the set of largest 3-event machines less than M, by successively eliminating each event. To eliminate event 0, since m^0 transitions to m^3 on event 0, these two states are combined. This is repeated for all states and the largest machine containing all the combined states self looping on event 0 is M_1. Similarly, the largest machines not acting on events 3,1 and 2 are M_2, M_3 and $M_\perp$ respectively. The *reduceEvent* algorithm returns M_1 and M_2 as the only incomparable machines in this set. The *eventDecompose* algorithm returns $\mathcal{E} = \{M_1, M_2\}$, since each pair of states in M are separated by M_1 or M_2. Hence, the 4-event M can be replaced by the 3-event M_1 and M_2, i.e., $\mathcal{E} = \{M_1, M_2\}$ is a (2,1)-event decomposition of M.

Theorem 8. Given machine M $(X_M, \alpha_M, \Sigma_M, m^0)$, the *eventDecompose* algorithm generates a (k,e)-event decomposition of M (if it exists) for some $k \leq |X_M|^2$.

Proof. The *reduceEvent* algorithm exhaustively generates the largest incomparable machines that ignore at least one event in Σ_M. After e such reduction in events, Loop 3 selects one machine (if it exists) among $\mathcal{M}$ to separate each pair of states in X_M. This ensures that at the end of Loop 3, either $d_{min}(\mathcal{E}) > 0$ or the algorithm has returned {} (no (k,e)-event decomposition

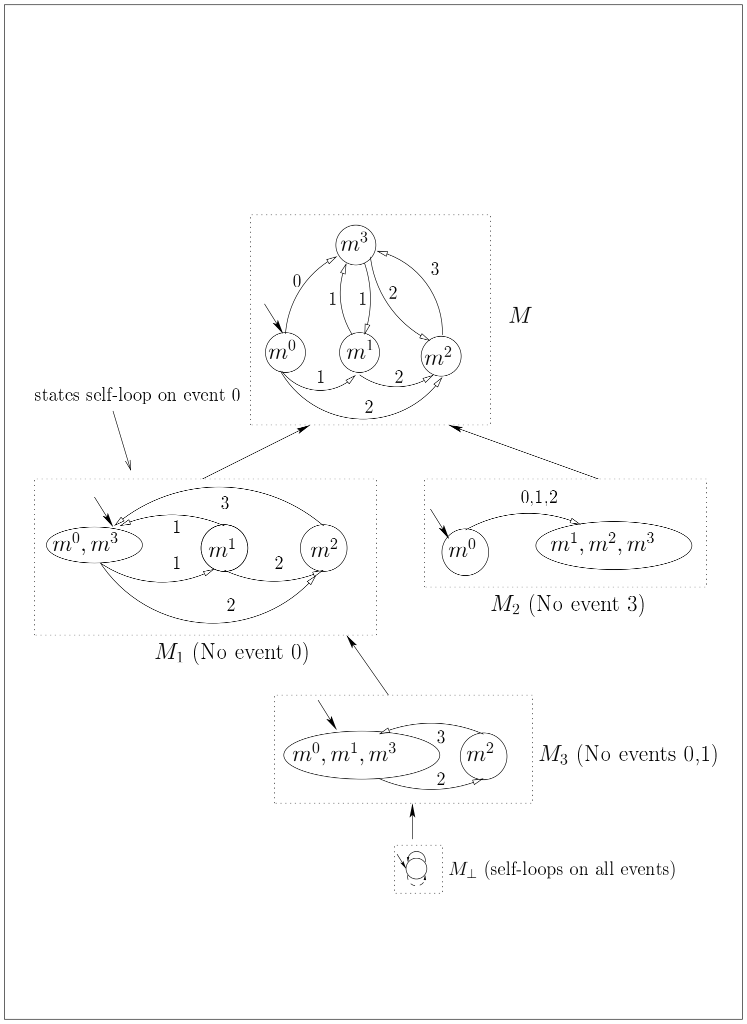


Figure 4.12: Example for the Event-based decomposition of a machine.

exists). Since there are at most $|X_M|^2$ pairs of states in X_M, there are at most $|X_M|^2$ iterations of Loop 3, in which we pick one machine per iteration. Hence, $k \leq |X_M|^2$. $\square$

The *reduceEvent* algorithm visits each state of machine M to create blocks of states that loop to the same block on event $\sigma \in \Sigma_M$. This has time complexity $O(|X_M|)$ per event. The cost of generating the largest closed partition corresponding to this block is $O(|X_M||\Sigma_M|)$ per event. Since we need to do this for all events in Σ_M, the time complexity to reduce at least one event is $O(|X_M||\Sigma_M|^2)$. In the *eventDecompose* algorithm, the first iteration generates at most $|\Sigma_M|$ machines, the second iteration at most $|\Sigma_M|^2$ machines and the e^{th} iteration will contain $O(|\Sigma_M|^e)$ machines. The rest of the analysis is similar to the one presented in section 4.2.2 and the time complexity of the *reduceEvent* algorithm is $O(|X_M||\Sigma_M|^{e+1})$.

To generate the (k,e)-event decomposition from the set of machines in $\mathcal{M}$, we find a machine in $\mathcal{M}$ to separate each pair of states in X_M. Since there are $O(|X_M|^2)$ such pairs, the number of iterations of Loop 3 is $O(|X_M|^2)$. In each iteration of Loop 3, we find a machine among the $O(|\Sigma_M|^e)$ machines of $\mathcal{M}$ that separates a pair $m_i, m_j \in X_M$. To check if a machine separates a pair of states just takes $O(|X_M|)$ time. Hence the time complexity of Loop 3 is $O(|X_M|^3|\Sigma_M|^e)$. So, the overall time complexity of the *eventDecompose* algorithm is the sum of the time complexities of Loop 1 and 3, which is $O(|X_M||\Sigma_M|^{e+1} + |X_M|^3|\Sigma_M|^e)$. So far, we have described our fusion-based approach for generating backup machines. In the following section, we focus on the algorithms for the detection and correction of faults in a fusion-based system.

```
detectByz
    Input: set of $f$ fusion states $B$, primary tuple $r$;
    Output: true if there is a Byzantine fault and false if not;
    for $(b \in B)$
        if $\neg(hash\_table(b) \cdot contains(r))$
            return true;
    return false;
```

Figure 4.13: Detection of Byzantine faults.

4.4 Detection and Correction of Faults

Consider a set of n primary machines and an $(f,\ f)$-fusion corresponding to it. In this section, we provide algorithms to detect Byzantine faults with time complexity $O(nf)$, on average, and correct crash/Byzantine faults with time complexity $O(n\rho f)$, with high probability, where n is the number of primaries, f is the number of crash faults and ρ is the average state reduction achieved by fusion. Throughout this section, we refer to Fig. 4.2, with primaries, $\mathcal{P} = \{A, B, C\}$ and backups $\mathcal{F} = \{F_1, F_2\}$, that can correct two crash faults. The execution state of the primaries is represented collectively as a n-tuple (referred to as the *primary tuple*) while the state of each backup/fusion is represented as the set of primary tuples it corresponds to (referred to as the *tuple-set*). In Fig. 4.2, if A, B, C and F_1 are in their initial states, then the primary tuple is $a^0b^0c^0$ and the state of F_1 is $f_1^0 = \{a^0b^0c^0, a^1b^0c^1, a^1b^1c^0, a^0b^1c^1\}$ (which corresponds to $\{r^0, r^2, r^4, r^5\}$).

4.4.1 Detection of Byzantine Faults

Given the primary tuple and the tuple-sets corresponding to the fusion states, the *detectByz* algorithm in Fig. 4.13 detects up to f Byzantine faults (liars). Assuming that the tuple-set of each fusion state is stored in a permanent hash table at the recovery agent, the *detectByz* algorithm simply checks if the primary tuple r is present in each backup tuple-set b. In Fig. 4.2, if the states of machines A, B, C, F_1 and F_2 are a^1, b^1, c^0, f_1^1 and f_2^1 respectively, then the algorithm flags a Byzantine fault, since $a^1b^1c^0$ is not present in either $f_1^1 = \{a^0b^1c^0, a^1b^1c^1, a^0b^0c^1, a^1b^0c^0\}$ or $f_2^1 = \{a^0b^1c^0, a^1b^0c^1\}$.

To show that r is not present in at least one of the backup tuple-sets in B when there are liars, we make two observations. First, we are only concerned about machines that lie within their state set. For example, in Fig. 4.2, suppose the true state of F_2 is f_2^0. To lie, if F_2 says it state is any number apart from f_2^1, f_2^2 and f_2^3, then that can be detected easily.

Second, like the fusion states, each primary state can be expressed as a tuple-set that contains the *RCP* states it belongs to. Immaterial of whether r is correct or incorrect, it will be present in all the truthful primary states. For example, in Fig. 4.2, if the correct primary tuple is $a^0b^0c^0$ then $a^0 = \{a^0b^0c^0, a^0b^1c^0, a^0b^1c^1, a^0b^0c^1\}$ contains $a^0b^0c^0$. If B lies, then the primary tuple will be $a^0b^1c^0$, which is incorrect. Clearly, a^0 contains this incorrect primary tuple as well.

Theorem 9. Given a set of n machines $\mathcal{P}$ and an (f, f)-fusion $\mathcal{F}$ corresponding to it, the *detectByz* algorithm detects up to f Byzantine faults among them.

Proof. Let r be the correct primary tuple. Each primary tuple is present in exactly one fusion state (the fusion states partition the *RCP* states), i.e, the correct fusion state. Hence, the incorrect fusion states (liars) will not contain r and the fault will be detected. If r is incorrect (with liars), then for the fault to go undetected, r must be present in all the fusion states.

If r^c is the correct primary tuple, then the truthful fusion states have to contain r^c as well, which implies that they contain $\{r, r^c\}$ in the same tuple-set. As observed above, the truthful primaries will also contain $\{r, r^c\}$ in the same tuple-set. So the execution state of all the truthful machines contain $\{r, r^c\}$ in the same tuple-set. Hence less than or equal to f machines, i.e, the liars, can contain r and r^c in distinct tuple-sets. This contradicts the fact that $\mathcal{F}$ is a (f, f)-fusion with greater than f machines separating each pair of RCP states. □

We consider the space complexity for maintaining the hash tables at the recovery agent. Note that, the space complexity to maintain a hash table is simply the number of points in the hash table multiplied by the size of each point. In our solution we hash the tuples belonging to the fusion states. In each fusion machine, there are N such tuples, since the fusion states partition the states of the RCP. Each tuple contains n primary states each of size $\log s$, where s is the maximum number of states in any primary. For example, $a^0b^1c^0$ in f_1^1 contains three primary states ($n = 3$) and since there are two states in A ($s = 2$) we need just one bit to represent it. Since there are f fusion machines, we hash a total of Nf points, each of size $O(n \log s)$. Hence, the space complexity at the recovery agent is $O(Nfn \log s)$.

Since each fusion state is maintained as a hash table, it will take $O(n)$ time (on average) to check if a primary tuple with n primary states is present in the fusion state. Since there are f fusion states, the time complexity for the *detectByz* algorithm is $O(nf)$ on average. Even for replication, the recovery agent needs to compare the state of n primaries with the state of each of its f copies, with time complexity $O(nf)$. In terms of worst-case message complexity, in fusion, we need to acquire the state of $n + f$ machines to detect the faults, while for replication, we need to acquire the state of $2nf$ machines.

4.4.2 Correction of Faults

Given a primary tuple r and the tuple-set of a fusion state, say b, consider the problem of finding the tuples in b that are within Hamming distance f of r. This is the key concept that we use for the correction of faults, as explained in sections 4.4.2.1 and 4.4.2.2. In Fig. 4.2, the tuples in $f_1^0 = \{a^0b^0c^0, a^1b^0c^1, a^1b^1c^0, a^0b^1c^1\}$ that are within Hamming distance one of a primary tuple $a^0b^0c^1$ are $a^0b^0c^0$, $a^1b^0c^1$ and $a^0b^1c^1$. An efficient solution to finding the points among a large set within a certain Hamming distance of a query point is *locality sensitive hashing* (LSH) [1, 35]. Based on this idea, we first select L hash functions $\{g_1 \dots g_L\}$ and for each g_j we associate an ordered set (increasing order) of k numbers C_j picked uniformly at random from $\{0 \dots n\}$. The hash function g_j takes as input an n-tuple, selects the coordinates from them as specified by the numbers in C_j and returns the concatenated bit representation of these coordinates. At the recovery agent, for each fusion state we maintain L hash tables, with the functions selected above, and hash each tuple in the fusion state. In Fig. 4.14 (i), g_1 and g_2 are associated with the sets $C_1 = \{0, 1\}$ and $C_2 = \{0, 2\}$ respectively. Hence, the tuple $a^1b^0c^1$ of f_1^0, is hashed into the 2^{nd} bucket of g_1 and the 3^{rd} bucket of g_2.

Given a primary tuple r and a fusion state b, to find the tuples among b that are within a Hamming distance f of r, we obtain the points found in the buckets $g_j(r)$ for $j = 1 \dots L$ maintained for b and return those that are within distance of f from r. In Fig. 4.14 (i), let $r = a^0b^1c^0$, $f = 2$, and $b = f_1^0$. The primary tuple r hashes into the 1^{st} bucket of g_1 and the 0^{th} bucket of g_2

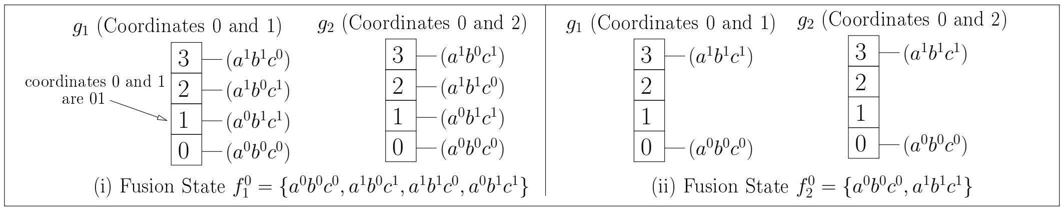

g_1 (Coordinates 0 and 1)
g_2 (Coordinates 0 and 2)
3
2
1
0
$(a^1b^1c^0)$
$(a^1b^0c^1)$
$(a^0b^1c^1)$
$(a^0b^0c^0)$
coordinates 0 and 1 are 01
$(a^1b^0c^1)$
$(a^1b^1c^0)$
$(a^0b^1c^1)$
$(a^0b^0c^0)$
(i) Fusion State $f_1^0 = \{a^0b^0c^0, a^1b^0c^1, a^1b^1c^0, a^0b^1c^1\}$
g_1 (Coordinates 0 and 1)
g_2 (Coordinates 0 and 2)
$(a^1b^1c^1)$
$(a^0b^0c^0)$
$(a^1b^1c^1)$
$(a^0b^0c^0)$
(ii) Fusion State $f_2^0 = \{a^0b^0c^0, a^1b^1c^1\}$

Figure 4.14: LSH example for fusion states in Fig. 4.2 with $k = 2$, $L = 2$.

which contains the points $a^0b^1c^1$ and $a^0b^0c^0$ respectively. Since both of them are withing Hamming distance two of r, both of the points are returned.

Assume a set of n-dimensional points P hashed using the LSH hash functions $\{g_1 \dots g_L\}$, where for each g_i, we associate k indices or positions as described above. Consider an n-dimensional query point q. We hash q with each of the L hash functions and return the points found in the buckets that are within a Hamming distance f of q, i.e, the f-neighbors of q. We now state and prove the main property of the LSH technique, as applicable to our work [1, 35].

Theorem 10. If we set $L = \log_{1-\gamma^k} \delta$, such that $(1 - \gamma^k)^L \leq \delta$, where $\gamma = 1 - f/n$, then each f-neighbor of q in P is returned with probability at least $1 - \delta$.

Proof. Consider two n-dimensional points a and b indexed by $[n]$. The probability that $a[i] = b[i]$, for some i in $[n]$, is equal to the fraction of coordinates on which a and b are the same. If the hamming distance between a and b is lesser than or equal to f, this probability is equal to $1 - f/n$.

Let $\mathcal{G}$ denote the probability that $g_i(a) = g_i(b)$ for some $i \in L$. In each hash function g_i, we pick k coordinates from the same positions from both a and b (positions selected uniformly at random). Hence, $\mathcal{G} = \prod_{i=1}^{i=k}$(probability

```
correctCrash
    **Input**: set of available fusion states $B$, primary tuple $r$,
    number of faults among the primaries $t$;
    **Output**: corrected primary $n$-tuple;
    $D \leftarrow \{\}$ //list of tuple-sets
    //find tuples in $b$ within Hamming distance $t$ of $r$
    **for** $(b \in B)$
        $S \leftarrow lsh\_tables(b) \cdot search(r, t)$;
        $D \cdot add(S)$;
    **return** Intersection of sets in $D$;
```

Figure 4.15: Correction of crash faults.

that $a[i] = b[i]) = (1 - f/n)^k$. Clearly, this probability is the same for all the hash functions in $\{g_1 \ldots g_L\}$.

The probability that an f-neighbor p of q is returned, denoted by $\mathcal{S}$, is the probability that $g_i(p) = g_i(q)$ for at least one value of $i \in L$. Hence, $\mathcal{S} = 1-(1-\mathcal{G})^L$. If we set $L = \log_{1-\mathcal{G}} \delta$, such that $(1-\mathcal{G})^L \leq \delta$, then $\mathcal{S} \geq 1-\delta$. The proof follows by substituting $\mathcal{G} = (1 - f/n)^k$. □

In the following sections, we present algorithms for the correction of crash and Byzantine faults based on the LSH functions.

4.4.2.1 Crash Correction

Given the primary tuple (with possible gaps because of faults) and the tuple-sets of the available fusion states, the *correctCrash* algorithm in Fig. 4.15 corrects up to f crash faults. The algorithm finds the set of tuple-sets S in each fusion state b, where each tuple belonging to S is within a Hamming distance t of the primary tuple r. Here, t is the number of faults among the primaries. To do this efficiently, we use the LSH tables of each fusion state. The set S returned for each fusion state is stored in a list D. If the intersection of the sets in D is singleton, then we return that as the correct primary tuple. If the intersection is empty, we need to exhaustively search each fusion state for points within distance t of r (LSH has not returned all of them), but this happens with a very low probability [1, 35].

In Fig. 4.2, assume crash faults in primaries B and C among $\{A, B, C\}$. Given the states of A, F_1 and F_2 as a^0, f_1^0 and f_2^0 respectively, the tuples within Hamming distance two of $r = a^0.\{empty\}.\{empty\}$ among $f_1^0 = \{a^0b^0c^0, a^1b^0c^1, a^1b^1c^0, a^0b^1c^1\}$ and $f_2^0 = \{a^0b^0c^0, a^1b^1c^1\}$ are $\{a^0b^0c^0, a^0b^1c^1\}$ and $\{a^0b^0c^0\}$ respectively. The algorithm returns their intersection, $a^0b^0c^0$ as the corrected primary tuple. In the following theorem, we prove that the *correctCrash* algorithm returns a unique primary tuple.

Theorem 11. Given a set of n machines $\mathcal{P}$ and an (f, f)-fusion $\mathcal{F}$ corresponding to it, the *correctCrash* algorithm corrects up to f crash faults among them.

Proof. Since there are t gaps due to t faults in the primary tuple r, the tuples among the backup tuple-sets within a Hamming distance t of r, are the tuples that contain r (definition of Hamming distance). Let us assume that the intersection of the tuple-sets among the fusion states containing r is not singleton. Hence all the available fusion states have at least two RCP states, $\{r^i, r^j\}$, that contain r. Similar to the proof in theorem 9, since both r^i and r^j contain r, these states will be present in the same tuple-sets of all the available primaries as well. Hence less than or equal to f machines, i.e, the failed machines, can contain r^i and r^j in distinct tuple-sets. This contradicts the fact that $\mathcal{F}$ is an (f, f)-fusion with greater than f machines separating each pair of RCP states. □

The space complexity analysis is similar to that for Byzantine detection since we maintain hash tables for each fusion state and hash all the tuples belonging to them. Assuming L is a constant, the space complexity of storage at the recovery agent is $O(Nfn \log s)$.

Let ρ be the average state reduction achieved by our fusion-based technique. Each fusion machine partitions the states of the RCP and the average size of each fusion machine is N/ρ. Hence, the number of tuples (or points) in each fusion state is ρ. This implies that there can be $O(\rho)$ tuples in each fusion state that are within distance f of r. So, the cost of hashing r and retrieving $O(\rho)$ n-dimensional points from $O(f)$ fusion states in B is $O(n\rho f)$ w.h.p (assuming k, L for the LSH tables are constants). So, the cost of generating D is $O(n\rho f)$ w.h.p. Also, the number of tuple sets in D is $O(\rho f)$.

In order to find the intersection of the tuple-sets in D in linear time, we can hash the elements of the smallest tuple-set and check if the elements of the other tuple-sets are part of this set. The time complexity to find the intersection among the $O(\rho f)$ points in D, each of size n is simply $O(n\rho f)$. Hence, the overall time complexity of the *correctCrash* algorithm is $O(n\rho f)$ w.h.p. Crash correction in replication involves copying the state of the copies of the f failed primaries which has time complexity $\theta(f)$. In terms of message complexity, in fusion, we need to acquire the state of all n machines that remain after f faults. In replication we just need to acquire the copies of the f failed primaries.

4.4.2.2 Byzantine Correction

Given the primary tuple and the tuple-sets of the fusion states, the *correctByz* algorithm in Fig. 4.16 corrects up to $\lfloor f/2 \rfloor$ Byzantine faults. The

```
correctByz
    Input: set of $f$ fusion states $B$, primary tuple $r$;
    Output: corrected primary $n$-tuple;
    $D \leftarrow \{\}$ //list of tuple-sets
    //find tuples in $b$ within Hamming distance $\lfloor f/2 \rfloor$ of $r$
    for ($b \in B$)
        $S \leftarrow lsh\_tables(b) \cdot search(r, \lfloor f/2 \rfloor)$;
        $D \cdot add(S)$;
    $G \leftarrow$ Set of tuples that appear in $D$;
    $V \leftarrow$ Vote array of size $|G|$;
    for ($g \in G$)
        // get votes from fusions
        $V[g] \leftarrow$ Number of times $g$ appears in $D$;
        // get votes from primaries
        for ($i = 1$ to $n$)
            if($r[i] \in g$)
                $V[g]++$;
    return Tuple $g$ such that $V[g] \geq n + \lfloor f/2 \rfloor$;
```

Figure 4.16: Correction of Byzantine faults.

algorithm finds the set of tuples among the tuple-sets of each fusion state that are within Hamming distance $\lfloor f/2 \rfloor$ of the primary tuple r using the LSH tables and stores them in list D. It then constructs a vote vector V for each unique tuple in this list. The vote for each tuple $g \in V$ is the number of times it appears in D plus the number of primary states of r that appear in g. The tuple with greater than or equal to $n + \lfloor f/2 \rfloor$ votes is the correct primary tuple. When there is no such tuple, we need to exhaustively search each fusion state for points within distance $\lfloor f/2 \rfloor$ of r (LSH has not returned all of them). In Fig. 4.2, let the states of machines A, B, C F_1 and F_2 be a^0, b^1, c^0, f_1^0 and f_2^0 respectively, with one liar among them ($\lfloor f/2 \rfloor = 1$). The tuples within Hamming distance one of $r = a^0b^1c^0$ among $f_1^0 = \{a^0b^0c^0, a^1b^0c^1, a^1b^1c^0, a^0b^1c^1\}$ and $f_2^0 = \{a^0b^0c^0, a^1b^1c^1\}$ are $\{a^0b^0c^0, a^1b^1c^0, a^0b^1c^1\}$ and $\{a^0b^0c^0\}$ respectively. Here, $a^0b^0c^0$ wins a vote each from F_1 and F_2 since $a^0b^0c^0$ is present in f_1^0 and f_2^0. It also wins a vote each from A and C, since the current states of A and C, a^0 and c^0, are present in $a^0b^0c^0$. The algorithm returns $a^0b^0c^0$ as the true primary tuple, since $n + \lfloor f/2 \rfloor = 3 + 1 = 4$. We show in the following theorem that the true primary tuple will always get sufficient votes.

Theorem 12. Given a set of n machines $\mathcal{P}$ and an (f, f)-fusion $\mathcal{F}$ corresponding to it, the *correctByz* algorithm corrects up to $\lfloor f/2 \rfloor$ Byzantine faults among them.

Proof. We prove that the true primary tuple, r^c will uniquely get greater than or equal to $(n + \lfloor f/2 \rfloor)$ votes. Since there are less than or equal to $\lfloor f/2 \rfloor$ liars, r^c will be present in the tuple-sets of greater than or equal to $n + \lfloor f/2 \rfloor$ machines. Hence the number of votes to r^c, $V[r^c]$ is greater than or equal to $(n + \lfloor f/2 \rfloor)$. An incorrect primary tuple r^w can get votes from less than or equal to $\lfloor f/2 \rfloor$ machines (i.e, the liars) and the truthful machines that contain both r^c and r^w in the same tuple-set. Since $\mathcal{F}$ is an $(f,\, f)$-fusion of $\mathcal{P}$, among all the $n + f$ machines, fewer than n of them contain $\{r^c, r^w\}$ in the same tuple-set. Hence, the number of votes to r^w, $V[r^w]$, is less than $(n + \lfloor f/2 \rfloor)$, which is less than $V[r^c]$. □

The space complexity analysis is similar to crash correction. The time complexity to generate D, same as that for crash fault correction is $O(n\rho f)$ w.h.p. If we maintain G as a hash table (standard hash functions), to obtain votes from the fusions, we just need to iterate through the f sets in D, each containing $O(\rho)$ points of size n each and check for their presence in G in constant time. Hence the time complexity to obtain votes from the backups is $O(n\rho f)$. Since the size of G is $O(\rho f)$, the time complexity to obtain votes from the primaries is again $O(n\rho f)$, giving an over all time complexity of $O(n\rho f)$ w.h.p. In the case of replication, we just need to obtain the majority across f copies of each primary with time complexity $O(nf)$. The message complexity analysis is the same as Byzantine detection, because correction can take place only after detection.

4.5 Comparison of Replication and Fusion

In this section, we summarize the main differences between replication and fusion (Table 4.2) for state machines. Throughout this section, we assume n primary machines, containing at most $O(s)$ states. We assume that the system can correct either f crash faults or f Byzantine faults.

Number of Backups To correct f crash faults among n primaries, replication requires nf backups while the *genFusion* algorithm in Fig. 4.6 generates

Table 4.2: Replication vs. Fusion for State Machines

	Replication Crash	**Fusion Crash**	**Replication Byzantine**	**Fusion Byzantine**
Number of Backups	nf	f	$2nf$	$2f$
Backup Space	s^{nf}	$(s^n/\rho)^f$	s^{2nf}	$(s^n/\rho)^{2f}$
Average Events/Backup	$\|\Sigma\|/n$	$\|\Sigma\|/\beta$	$\|\Sigma\|/n$	$\|\Sigma\|/\beta$
Backup Generation Time Complexity	$O(nsf)$	$O(s^n\|\Sigma\|f/\rho^n)$	$O(nsf)$	$O(s^n\|\Sigma\|f/\rho^n)$
Fault Detection Time	$O(1)$	$O(1)$	$O(nf)$	$O(nf)$ on avg.
Fault Correction Time	$O(f)$	$O(n\rho f)$ w.h.p	$O(nf)$	$O(n\rho f)$ w.h.p
Fault Detection Messages	$O(1)$	$O(1)$	$2nf$	$n + f$
Fault Correction Messages	f	n	$2nf$	$n + f$

f backups. To correct f Byzantine faults, fusion requires $2f$ backups as compared to the $2nf$ backups required by replication.

Backup Space In replication we maintain nf additional backup copies each of size $O(s)$ and hence the total state space is $O(s^{nf})$. Given n primaries each with $O(s)$ states, their RCP has $O(s^n)$ states. Since the average state reduction is ρ, the size of each fusion is $O(s^n/\rho)$. Hence the total backup state space in fusion is $O(s^n/\rho)^f$. The only difference for Byzantine faults is that we maintain $2f$ backups for fusion and $2nf$ backups for replication.

Average Events/Backup If the union of the event sets of all primary machines is denoted by Σ, then the average number of events in each primary copy is $|\Sigma|/n$. Hence, for replication, the average number of events per backup is $|\Sigma|/n$. In the case of fusion, we assume that the average event reduction achieved over the RCP is β. Since the RCP contains $|\Sigma|$ events, the average events per backup in fusion is $|\Sigma|/\beta$. Note that, this is one of the major trade-offs between replication and fusion. For small values of β, the number of events in each backup can be much larger than replication, leading to increased load on the backup servers.

Backup Generation Time Complexity In replication, we just generate nf copies with time complexity $O(nsf)$. The time complexity to generate fusions, as shown in section 4.2.3 is $O(s^n|\Sigma|f/\rho^n)$. The results are similar for Byzantine faults. Since the time complexity to generate backups in fusion is far more than in replication, this is the other major trade-off between replication and fusion.

Detection and Correction of Faults The explanation for the remaining rows in the table can be found in section 4.4. For small values of $n\rho$, fusion causes almost no overhead over replication in terms of the time complexity for the detection and correction of faults. Also, fusion causes more message overhead for recovery in crash faults, while it causes lesser overhead for Byzantine faults.

Chapter 5

Fused Data Structures

In this chapter, we present the design of fused data structures and the theoretical properties guaranteed by them. Given a set of n different data structures, we correct f crash faults using just f backups as compared to the nf backups required by replication. In addition, we ensure $O(n)$ savings in space over replication. We present the design of fused backups for most commonly used data structures such as lists, stacks, vectors and hash tables. Further, we correct f Byzantine faults using just $nf + f$ backups as compared to the $2nf$ backups required by replication. Our theoretical results illustrate that the fused backups are space-efficient as compared to replication while they cause very little additional overhead for normal operation. In the following section, we extend the model presented in section 1.3 to focus on aspects specific to data structures.

5.1 Model and Notation

Our system consists of independent distributed servers hosting data structures. We denote the n given data structures, also referred to as primaries, $X_1 \ldots X_n$. The backup data structures that are generated based on our idea of fusing primary data are referred to as *fused backups* or *fused data structures*. The operator used to combine primary data is called the *fusion operator*. The number of fused backups, t, depends on the fusion operator and the number of faults that need to be corrected. The fused backups are denoted $F_1 \ldots F_t$. For example, when XOR/sum is the fusion operator, we maintain one fused backup.

The data structures are modeled as a set of data nodes and an index structure that specifies order information about these data nodes. For example, the index structure for a linked list includes the head, tail and next pointers. The data structures in our system have a *state* as well as an *output* associated with them. The state of a data structure is a snapshot of the values in the data nodes and the index structure. The output is the value visible to the external world or client. On application of an event/update, the data structure transitions from one state to another and changes its output value. For example, the state associated with a linked list is the value of its nodes, next pointers, tail and head pointers. When we insert data into a linked list with a certain key, the value of the nodes and pointers change (state) and it responds with either success or failure (output). We make two assumptions regarding the primary data structures in our system:

1. The size of data in the data structure far exceeds the size of its index structure. This is usually true for most practical applications in which the data nodes are in the order of megabytes while the index structures are in the order of bytes (for example, next pointers in a linked list).

2. The data nodes in the data structures are completely uncorrelated and cannot be compressed further.

In the following section, we present the design of fused backups. The design of the fused data structures is independent of the fault model and for simplicity we explain the design assuming only crash faults. Henceforth, when we simply say faults, we refer to crash faults.

5.2 Fusion-based Fault Tolerant Data Structures

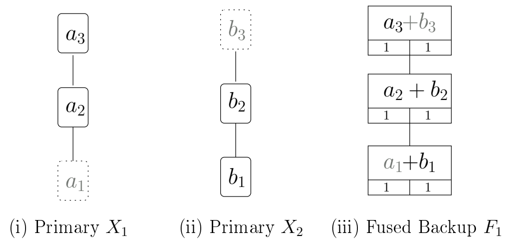


Figure 5.1: Old Fusion [34]

Design Motivation In [34], the authors present a design to fuse array and list-based primaries that can correct one crash fault. We highlight the main drawback of their approach for linked lists. The fused structure for linked list primaries in [34] is a linked list whose nodes contain the XOR (or sum) 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 element is inserted in the correct position at the backup by iterating through the fused nodes using the bit array; a similar operation is performed for deletes. An example is shown in Fig. 5.1 with two primaries and one backup. After the deletion of primary elements a_1 and b_3, the first and third nodes of the fused backup F_1 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 provide a generic design of fused backups, for all types of data structures such that the fused backup contains only as many nodes as the largest primary, while guaranteeing efficient updates? We present a solution for linked lists and then generalize it for complex data structures.

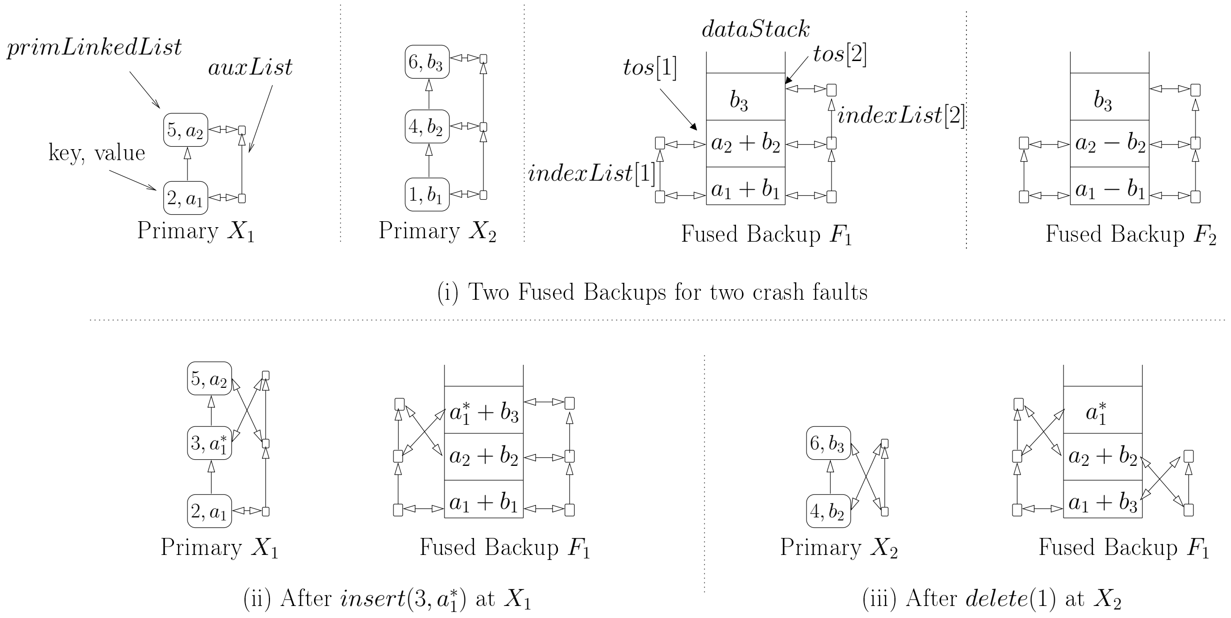


Figure 5.2: Fused Backups for Linked Lists (Keys not shown in F_1, F_2 due to space constraint)

5.2.1 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. Fig. 5.2 shows two primary sorted 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. At each fused backup, we also maintain index structures that replicate the ordering information of the primaries. The index structure corresponding to primary X_i is identical in structure to X_i, but while X_i consists of data nodes, the index structure only contains pointers to the fused nodes. The savings in space is achieved because primary nodes are being fused, while updates are efficient since we maintain the index structure of each primary at the backup.

We begin with a high-level description of how we restrict the number of nodes in the backup stack. At each backup, elements of primary X_i are simply inserted one on top of the other in the stack with a corresponding update to the index structure to preserve the actual ordering information. The case of deletes is more complex. If we just delete the element at the backup, then similar to Fig. 5.1, a 'hole' is 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 primary with the most number of nodes.

Since the top-most 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. Fig. 5.2 illustrates these operations with an example. We explain them in greater detail in the following paragraphs.

```
Insert at Primaries $X_i$ :: $i = 1..n$
    Input: key $k$, data value $d$;
    Var: Linked List $primaryLinkedList$ (given data
    structure), $auxList$ (order of data at each backup);
    if ($primaryLinkedList \cdot contains(k)$)
        /* key present, just update its value*/
        $old = primaryLinkedList \cdot get(k) \cdot value$
        $primaryLinkedList \cdot update(k, d)$;
        $send(k, d, old)$ 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 */
        $auxList.insertAtEnd(a)$;
        $primaryLinkedList \cdot insert(k, p)$;
        $send(k, d, null)$ to all fused backups;
```

Figure 5.3: Primary Design: Inserts

Inserts Fig. 5.3 and 5.4 show the algorithms 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*)

```
Insert at Fused Backups $F_j$ :: $j = 1..t$
    Input: key $k$, new value $d_i$, old value $old_i$;
    Var: Stack $dataStack$ (stack of fused nodes),
    Linked Lists[] $indexList[n]$ (order of primary data),
    Pointer to Fused Nodes[] $tos[n]$ (top of stack pointers);
    if ($indexList[i] \cdot contains(k)$)
        fusedNode $f = indexList[i] \cdot get(k)$;
        $f \cdot updateCode(old_i, d_i)$;
    else
        fusedNode $p = tos[i] + +$;
        if ($p == null$)
            $p$ = new fusedNode;
            $dataStack \cdot push(p)$;
        $p \cdot updateCode(0, d_i)$;
        $p \cdot refCount + +$;
        /* mimic primary linked list */
        indexNode $a$ = new indexNode;
        $a \cdot fusedNode = p$;
        $p \cdot indexNode[i] = a$;
        $indexList[i] \cdot insert(k, a)$;
```

Figure 5.4: Fused Backup Design: Inserts

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 (*dataStack*) 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 containing 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.

Fig. 5.2(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 F_1, the value of the second node (nodes numbered from zero) is updated to $a_1^* + b_3$ and a pointer to this node is inserted at index 1 of $indexList[1]$. The identical operation is performed at F_2 (not shown in the figure due to space constraints), with the only difference being that the second fused node is updated to $a_1^* - b_3$. Observe that the aux list at X_1 specifies the exact order of elements maintained at the backup stack $(a_1 \rightarrow a_2 \rightarrow a_1^*)$. Analogously, $indexList[1]$ 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 at Primaries $X_i$ :: $i = 1..n$
    Input: key $k$;
    Var: Linked List $primaryLinkedList$ (given data
    structure), $auxList$ (order of data at each backup);
    $p = primaryLinkedList \cdot delete(k)$;
    $old = p \cdot value$;
    /* 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 $a$ */
    $(a \cdot prev) \cdot next = auxTail$;
    $auxTail \cdot next = a \cdot next$;
    $delete\ a$;
```

Figure 5.5: Primary Design: Deletes

```
$Delete\ at\ Fused\ Backups\ F_j :: j = 1..t$
    **Input**: key $k$, old value $old_i$, end value $tos_i$;
    **Var**: Stack $dataStack$ (stack of fused nodes),
    Linked Lists[] $indexList[n]$ (order of primary data),
    Pointer to Fused Nodes[] $tos[n]$ (top of stack pointers);
    /* update fused node containing $old_i$
        with primary element of $X_i$ at $tos[i]$*/
    indexNode $a = indexList[i] \cdot delete(k)$;
    fusedNode $p = a \cdot fusedNode$;
    $p \cdot updateCode(old_i, tos_i)$;
    $tos[i] \cdot updateCode(tos_i, 0)$;
    $tos[i] \cdot refCount - -$;
    /* update index node pointing to $tos[i]$ */
    $tos[i] \cdot indexNode[i] \cdot fusedNode = p$;
    **if** ($tos[i].refCount == 0$)
        $dataStack.pop()$;
    $tos[i] - -$;
```

Figure 5.6: Fused Backup Design: Deletes

Deletes Fig. 5.5 and 5.6 show 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.

At the backup, since $indexList[i]$ preserves the exact order information of X_i, by a simple double dereference, we can obtain the fused node p that contains the element of X_i associated with k. The value of p is updated with the top-most element (sent by the primary as *tos*) to simulate the shift. The pointers of $indexList[i]$ are updated to reflect this shift. Figure 5.2 (*iii*) shows the state of X_1 and F_1 after the deletion of b_1. The key facts to note are: (*i*) at F_1, b_3 has been shifted from the end to the 0^{th} node (*ii*) the aux list at X_2 reflects the correct order of its elements at the backup stack ($b_3 \rightarrow b_2$) and (*iii*) $indexList[2]$ reflects the correct order of elements at X_2 ($b_2 \rightarrow b_3$).

As specified in section 5.1, we assume that the size of the data far exceeds the size of the index structure. This assumption extends to the auxiliary structures which are also in the order of bytes. So the space overhead of maintaining these auxiliary/index structures is negligible. Also, the auxiliary structures at the primary can be updated in constant time for both inserts and deletes with the use of double-ended pointers. Hence, they do not cause any additional overhead in terms of time. In the following section, we modify the algorithms in Fig. 5.4 and 5.6 to enable concurrent updates at the fused backups, by threads belonging to different primaries.

5.2.2 Concurrent Updates

Since the primaries are independent of each other, in many cases the updates to the backup can be to different fused nodes. We present an algorithm for concurrent updates in which we never have to lock the entire data stack at the fused backups. This can achieve considerable speed-up during normal operation.

Consider the algorithms shown in Fig. 5.7 and 5.8. We assume that all the operations on the fused nodes are atomic. The only difference in the algorithm for concurrent updates is the **lock-unlock** commands for some sections of the algorithm. This simply means that the lines of the algorithm (or

```
Inserts at Fused Backups $F_j :: j = 1..t$
    Input: key $k$, new value $d_i$, old value $old_i$;
    Var: Stack $dataStack$ (stack of fused nodes),
    Linked Lists[] $indexList[n]$ (order of primary data),
    Pointer to Fused Nodes[] $tos[n]$ (top of stack pointers);
    if $(indexList[i] \cdot contains(k))$
        fusedNode $f = indexList[i] \cdot get(k)$;
        $f \cdot updateCode(old_i, d_i)$;
    else
        lock{
            fusedNode $p = tos[i]++$;
            if $(p == null)$
                $p$ = new fusedNode;
                $dataStack \cdot push(p)$;
        }unlock;
        $p \cdot updateCode(0, d_i)$;
        $p \cdot refCount++$;
        /* mimic primary linked list */
        indexNode $a$ = new indexNode;
        $a \cdot fusedNode = p$;
        $p \cdot indexNode[i] = a$;
        $indexList[i] \cdot insert(k, a)$;
```

Figure 5.7: Concurrent Fused Backup Design: Inserts

program) within these sections are executed atomically. The atomicity for both the fused nodes and these lock-unlock sections can be implemented using standard locks.

Since we focus on concurrency among primary threads of distinct primaries, we do not lock any of the operations involving the index lists/index nodes. Further, the operations on the fused nodes are atomic. Hence, we only need to lock the sections of the program that change the data stack at the fused backup, i.e., lines that add nodes to the data stack or delete nodes from the data stack. This is reflected in the algorithms shown in Fig. 5.7 and 5.8,

```
Deletes at Fused Backups $F_j$ :: $j = 1..t$
    Input: key $k$, old value $old_i$, end value $tos_i$;
    Var: Stack $dataStack$ (stack of fused nodes),
    Linked Lists[] $indexList[n]$ (order of primary data),
    Pointer to Fused Nodes[] $tos[n]$ (top of stack pointers);
    /* update fused node containing $old_i$
        with primary element of $X_i$ at $tos[i]$*/
    indexNode $a = indexList[i] \cdot delete(k)$;
    fusedNode $p = a \cdot fusedNode$;
    $p \cdot updateCode(old_i, tos_i)$;
    $tos[i] \cdot updateCode(tos_i, 0)$;
    $tos[i] \cdot refCount--$;
    /* update index node pointing to $tos[i]$ */
    $tos[i] \cdot indexNode[i] \cdot fusedNode = p$;
    lock{
        if ($tos[i].refCount == 0$)
            $dataStack.pop()$;
    }unlock;
    $tos[i]--$;
```

Figure 5.8: Concurrent Fused Backup Design: Deletes

where we ensure that the sections of the algorithms that push nodes and pop nodes from the data stack are executed atomically. Hence, with just a few lines of sequential execution, we allow high levels of concurrency at the fused backups. In the following section, we extend our solution for linked lists and present a generic solution for more complex data structures.

5.2.3 Fused Backups for Complex Data Structures

The design of fused backup for linked lists can be generalized for most commonly used data structures. We explain this using the example of balanced binary search trees (BBST). Fig. 5.9 (i) shows two primary BBSTs and a fused backup. For simplicity, we explain the design using just one backup that can correct one crash fault. The index 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 are similar 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 a BBST containing the order information rather than a list. The update to the backup stack is identical to that of linked list primaries. Fig. 5.9 (ii) shows the state of X_1 and F_1 after the deletion of a_3 followed by the insertion 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. Similarly, $indexBBS[1]$ maintains the order of the elements at X_1. For example, as the root at X_1 contains a_1, the root of $indexBBST[1]$ points to the fused node containing a_1.

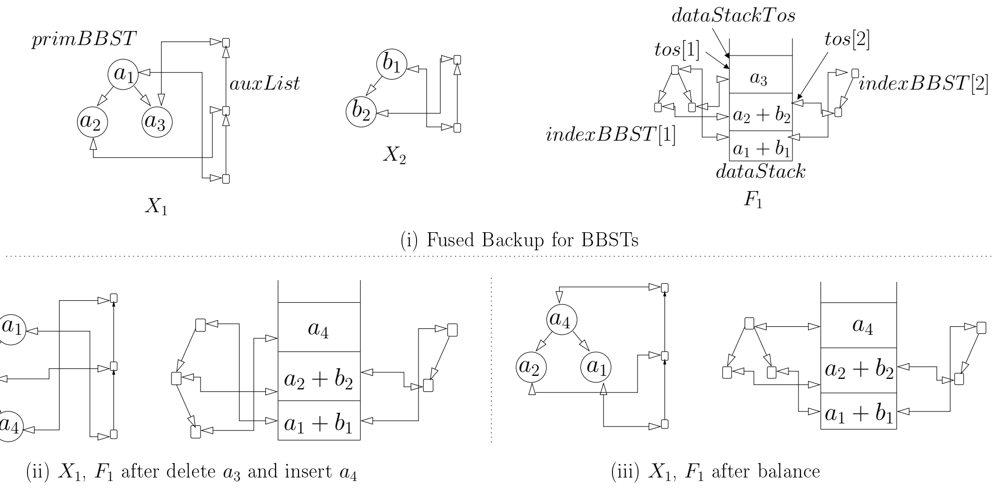


Figure 5.9: Fused Backups for Balanced Binary Search Trees (Keys not shown due to space constraint)

So far we have focused only on the insert and delete operations to the data structure, since those are the operations that add and delete data nodes. However, since we maintain the entire index structure at the backups, we support all operations that do not involve decoding the values in the fused nodes of the backup. We illustrate this with the example of the balance operation in the BBST shown in Fig. 5.9 (*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 index 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 design, we have implemented fused backups for linked lists, vectors, queues, hash tables and tree maps. So far we have only focused on one or two crash faults. In the following section, we extend this solution, using Reed-Solomon (RS) codes [79], to correct any number of faults.

5.2.4 Reed-Solomon Codes for f faults

In section 3.2.3.1 we explain the basics of RS coding. In this section, we focus on its use as a fusion operator for the fused data structures. Let the n data words corresponding to n data nodes of the respective primaries be denoted by $D = \{d_1, d_2, \dots d_n\}$. RS coding generates f checksum/code words $\{c_1, c_2, \dots c_f\}$ that can correct f erasures among the data and the checksum words. The f fused nodes at the same position in each of the f fused backups contain these checksum words. Hence, we correct f faults among the primaries using f backups.

Let the encoding/information dispersal matrix for the code be denoted by B. Let 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 checksum words (fused data) computed for the data words in D (primary data). In the following paragraphs, we focus on the fusion operations that use the RS coding routines.

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 Fig. 5.4 and 5.6.

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 satisfy to the equation, $D \times B' = P'$. When exactly f data words fail, B' is a $n \times n$ matrix. The data words can be generated as follows: $P' \times (B')^{-1} = D$.

In Fig. 5.2, we have used simple sum-difference as the fusion operator that can correct two crash faults. We now present an example using RS codes to correct three crash faults. Here, we maintain three fused backups F_1, F_2 and F_3, each of identical structure but with different values in the fused nodes. We first generate the information dispersal matrix $B = \begin{bmatrix} I & S \end{bmatrix}$ for $n = 3$, $f = 3$ (explained in section 3.2.3.1).

In Fig. 5.2, consider the fused node in the 0^{th} position in F_1 and F_2, that contain the sum and difference of the primary elements a_1 and b_1. For RS codes, we first generate the checksum blocks, $C = \begin{bmatrix} c_1 & c_2 & c_3 \end{bmatrix}$ for $D = \begin{bmatrix} a_1 & b_1 \end{bmatrix}$. The fused nodes in the 0^{th} position of F_1, F_2 and F_3 will contain the values c_1, c_2 and c_3 respectively. In Fig. 5.2 (ii), when an element is inserted into X_1, the 2^{nd} node of F_1 and F_2 is updated to $a_1^* + b_3$ and $a_1^* - b_3$ respectively. With RS codes as the fusion operator, the code value of the 2^{nd} node in the fused backup F_j $(j : 1 \ldots 3)$ is updated with [old code value + $(b_{j,1})(a_1^*)$].

So far, we have presented the design of fused backups for commonly used data structures to correct f faults among the primaries. In the following section we present the theoretical properties of these fused backups.

5.3 Theory of Fused Data Structures

In this section we prove properties on the fused backups such as size optimality, update efficiency and update order independence, 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 primary data structures 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 updates in RS codes).

5.3.1 Space Optimality

Consider n primaries, each containing $O(m)$ nodes, each of size $O(s)$. In [34], to correct one crash fault, the backup for linked lists and list-based queues requires $O(nms)$ space, which is as bad as replication. We show that the fused backups presented in this dissertation require only $O(ms)$ space. Further, to correct f faults, we show that the fused backups need only $O(msf)$ space. Replication, on the other hand requires $O(mnsf)$ space, which is $O(n)$ times more than fusion. To correct f crash faults, we use RS codes that require f fused backups, which is the minimum number of backups required for f faults. For example, in Fig. 5.2, the number of fused nodes in F_1 or F_2 is always equal to the number of nodes in the largest primary. The optimal size of the data stack in our backups combined with RS codes as the fusion operator, leads to the result that our solution is space optimal when the data across the primaries is uncorrelated and incompressible.

Theorem 5.3.1 (Space Optimality)**.** *The fused backups generated by our design using RS codes as the fusion operator are of optimal size.*

Proof. We first show that the data stack of each backup contains only m fused nodes. A *hole* is defined as a fused node that does not contain an element from a primary followed by a fused node that contains an element from that primary. When there are no holes in the data stack, each primary element is stacked one on top of the other and the stack contains only m nodes i.e as many nodes as the largest primary. We maintain the invariant that our data stack never has holes.

In inserts to X_i, we always update 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 by $tos[i]$ to plug this hole. So, each fused backup contains at most m nodes.

If the size of each node is $O(s)$, then the backup space required by our solution to correct f crash faults is $O(msf)$. Now, f crash faults among the primaries will result in the failure of at least $O(mf)$ data nodes, each of size $O(s)$. Hence, any solution for fault tolerance requires $O(msf)$ space. □

5.3.2 Efficient Updates

In [34], to update the backup for linked lists, we may have 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 $O(nm)$. However, since each primary linked list has $O(m)$ nodes, the update to a primary takes only $O(m)$ time. Hence the solution in [34] is not update efficient.

We show that updates to the fused backups presented in this dissertation take only as much time as that at the corresponding primary. So, fusion causes minimal overhead during normal operation as compared to replication. Our proof is based on the following simple intuition. The time complexity of update to the primaries depends on its index structure. For example, in the case of a linked list the index structure consists of next pointers. So to update a linked list with $O(m)$ nodes it takes $O(m)$ time. Since we replicate the index structure of each primary completely at the backup, the time complexity of the update to the fused backup is same as that at the primary.

Theorem 5.3.2 (Update Efficieny)**.** *The time complexity of the updates to a fused backup is of the same order as that at the primary.*

Proof. 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 index structure consists of an insertion 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 index 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. Shifting the final element of this primary to the fused node that contains the deleted element is done in constant time.

This argument for inserts and deletes extends to more complex operations: any operation performed on the primary will also be performed on the index structure at the backup. Updating the data nodes of the stack takes constant time. $\square$

5.3.3 Order Independence

In the absence of any synchronization at the backups, updates from different primaries can be received in any order at the backups. The assumption of FIFO communication channels only guarantees that the updates from the *same primary* will be received by all the backups in the same order. A direct extension of the solution in [34] for multiple faults can result in a state from which recovery is impossible. For example, in Fig. 5.2, F_1 may receive the insert to X_1 followed by the delete to X_2 while F_2 may receive the delete update followed by the insert. To achieve recovery, it is important that the fused nodes at the same position at different fused backups contain the same primary elements (in different coded forms). In Fig. 5.2 (i), if the 0^{th} node of F_1 contains $a_1 + b_1$, while the 0^{th} node of F_2 contains $a_2 - b_1$, then we cannot recover the primary elements when X_1 and X_2 fail.

We show that in the current design of fused backups, the nodes in the same position across the fused backups always contain the same primary elements independent of the order in which the updates are received at the backups. Also, the index structures at the backups are also independent of the order in which the updates are received. Consider the updates shown in Fig. 5.2. The updates to the index lists 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 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.

Theorem 5.3.3 (Order Independence/Commutativity)**.** *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 index structure commute. As far as updates to the stack are concerned, the proof follows from two facts about our design. First, updates on the backup for a certain primary do not affect the order of elements of the other primaries at the backup. Second, the state of the backup after an update from a primary depends only on the order of elements of that primary. The same argument extends to other complex operations that only affect the index structure. □

5.3.4 Fault Tolerance with Limited Backup Servers

So far we have implicitly assumed that the primary and backup structures reside on independent servers for the fusion-based solution. 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. Consider a set of n data structures, each residing on a distinct server. We need to correct f crash faults among the servers given only γ additional servers to host the backup structures. We present a solution to this problem that requires $\lceil n/(n+\gamma-f)\rceil f$ backups and show that this is the necessary and sufficient number of backups for this problem. Further, we present an algorithm for generating the optimal number of backups.

To simplify our discussion, we start with the assumption that *no* additional servers are available for hosting the backups ($\gamma = 0$). 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 ($n = 5$, $f = 3$). In a direct fusion-based solution, we will just generate three backups F_1, F_2, F_3, and distribute them among any three servers, say, H_1, H_2 and 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 f backups obtained by fusing the primaries $X_{i_1}, X_{i_2}, \ldots, X_{i_t}$, by $F_1(i_1, i_2, \ldots, i_t), F_2(i_1, i_2, \ldots, i_t) \ldots F_f(i_1, i_2, \ldots, i_t)$. For example, 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:

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

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

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

Note that, 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.

For example, assume crash faults in the servers H_2, H_4 and H_5. Consider the recovery of X_2 on the crashed server, H_2. Since, $F_1(1,2)$, $F_2(1,2)$, $F_3(1,2)$ are the three fused backups for $[X_1, X_2]$, given the state of any two data structures among $\{X_1, X_2, F_1(1,2), F_2(1,2), F_3(1,2)\}$, we can recover the state of the remaining three. In our example, we can obtain the state of X_1 on server H_1, and the state of $F_1(1,2)$ on server H_3 (servers that have not crashed). Given the state of these two data structures we can recover the state of X_2, $F_2(1,2)$ and $F_3(1,2)$. 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 an algorithm to correct f faults among the servers.

Partitioning Algorithm: Partition the set of n primaries, each residing on a distinct server as evenly as 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 $\lceil n/(n-f) \rceil f$. Replication, on the other hand requires nf backup structures which is always greater than or equal to $\lceil n/(n-f) \rceil f$. We show that $\lceil n/(n-f) \rceil f$ 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 at least 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. We generalize this result in the following theorem.

Theorem 5.3.4. *Given a set of n data structures, each residing on a distinct server, to correct f crash faults among the servers, it is necessary and sufficient to add $\lceil n/(n+\gamma-f)\rceil f$ backup structures, when there are only γ additional servers available to host the backup structures.*

Proof. We first prove sufficiency, followed by the proof showing that it is necessary to maintain that many backups.

(Sufficiency): We modify the partitioning algorithm for γ additional servers simply by partitioning the primaries into $\lceil n/(n+\gamma-f)\rceil$ blocks rather than $\lceil n/(n-f)\rceil$ blocks. Since the maximum number of primaries in any block of the partitioning algorithm is $n+\gamma-f$, 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 $t < \lceil n/(n+\gamma-f) \rceil f$. 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,

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

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 $n + \gamma - f$ 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, $\lceil f/(n+\gamma-f) \rceil$. Since, $(t-f)/f < \lceil f/(n+\gamma-f) \rceil$, 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. $\square$

Minimality We now define a partial order among equal sized sets of backups and prove that the partitioning algorithm generates a *minimal* set of backups. Given a set of four data structures, $\{X_1 \ldots X_4\}$, each residing on a distinct server, consider the problem of correcting two faults among the servers, with no additional backup servers ($n = 4$, $f = 2$, $\gamma = 0$). Since, $\lceil n/(n+\gamma-f) \rceil = 2$, the partitioning algorithm will partition the set of primaries into two blocks, say $[X_1, X_2]$ and $[X_3, X_4]$ and generate four fused backups, $F_1(1, 2)$, $F_2(1, 2)$ and $F_1(3, 4)$, $F_2(3, 4)$. An alternate solution to the problem is to fuse the entire set of primaries to generate four fused backups, $F_1(1, 2, 3, 4) \ldots F_4(1, 2, 3, 4)$. Here, $F_1(1, 2)$ is obtained by fusing the primaries X_1 and X_2, whereas $F_1(1, 2, 3, 4)$ is obtained by fusing all four primaries. In the latter case, maintenance is more expensive, since the backups need to receive and act on updates corresponding to all the primaries, whereas in the former, each backup receives inputs corresponding to just two primaries. Based on this idea, we define an order among backups.

Given a set of n data structures, X, consider backups F and F', obtained by fusing together a set of primaries, $M \subseteq X$ and $N \subseteq X$ respectively. F is less than F' ($F < F'$) if $M \subsetneq N$. In the example discussed, $F_1(1, 2) < F_1(1, 2, 3, 4)$, as $\{X_1, X_2\} \subsetneq \{X_1, X_2, X_3, X_4\}$. We extend this to define an order among sets of backups that correct f faults among the servers.

Definition 17. (Order among Sets of Backups) Given a set of n data structures, each residing on a distinct server, consider two sets of t backups, Y and Y' that correct f faults among the servers. Y is less than Y', denoted $Y < Y'$, if the backups in Y can be ordered as $\{F_1, ..F_t\}$ and the backups is Y' can be ordered as $\{F'_1, ..F'_t\}$ such that $(\forall 1 \leq i \leq t : F_i \leq F'_i) \wedge (\exists j : F_j < F'_j)$.

A set of backups Y is *minimal* if there exists no set of backups Y' such that $Y' < Y$. In the example for $n = 4$, $f = 2$, the set of backups, $Y = \{F_1(1, 2), F_2(1, 2), F_1(3, 4), F_2(3, 4)\}$, generated by the partitioning algorithm is clearly less than the set of backups, $Y' = \{F_1(1, 2, 3, 4) \ldots F_4(1, 2, 3, 4)\}$. We show that the partitioning algorithm generates a minimal set of backups.

Theorem 5.3.5. *Given a set of n data structures, each residing on a distinct server, to correct f faults among the servers, the partitioning algorithm generates a minimal set of backups.*

Proof. When a backup F is generated by fusing together a set of primaries, we say that each primary in the set *appears* in the backup. Given a set of backups that can correct f faults among the servers, each primary has to appear at least f times across all the backups. The partitioning algorithm generates a set of backups Y_p, in which each primary appears exactly f times. Any other solution in which the primaries appear exactly f times will be incomparable to Y_p. $\square$

In the following section, we present the algorithms for the detection and correction of faults.

5.4 Detection and Correction of Faults

Given n primary data structures, each with $O(m)$ nodes of size $O(s)$ each, we present an algorithm to correct f crash faults with time complexity $O(nmsf^2)$. Further, we present a solution to detect and correct f Byzantine faults using just $nf + f$ backups that causes no additional overhead during normal operation. The time complexity of recovery in Byzantine faults is $O(msft^2 + nst^2)$, where t is the actual number of liars in the system.

5.4.1 Crash Faults

To correct crash faults, the client needs to accquire the state of all the available data structures, both primaries and backups. As seen in Section

5.2, the fused node at the same position at all the fused backups contains the codeword 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. The decoding algorithm depends on the erasure code used. In Fig. 5.2 (i), to recover the state of the failed primaries, we obtain the state 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. Note that, even though we have explained the recovery algorithm mainly for crashed primaries, this can be easily extended to faults among the fused backups.

We consider the time complexity of recovery using RS codes as the fusion operator. Given n data values, the cost of recovering f values, each of size s by RS decoding is $O(nsf^2)$ [74]. 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 $O(nmsf^2)$ where each primary has $O(m)$ nodes of $O(s)$ size each. Recovery is much cheaper in replication and has time complexity $O(msf)$.

5.4.2 Byzantine Faults

To correct f Byzantine faults among n primaries pure replication requires $2f$ additional copies of each primary, which ensures that a non-faulty majority of $f + 1$ copies are always available. Hence, the correct state of the data structure can easily be ascertained. This approach requires $2nf$ backup data structures in total. In this section, we present a hybrid solution that combines fusion with replication to correct f Byzantine faults using just $nf + f$ backup structures, while ensuring minimal overhead during normal operation. However, the time complexity for recovery in fusion is $O(msft^2 + nst^2)$, whereas it is only $O(mst)$ for replication, where m is the number of nodes in each data structure, s is the size of each node and t is the actual number of Byzantine faults. In a system with infrequent faults, this maybe an acceptable compromise for the savings in the number of backups.

In [33], the author presents a fusion-based algorithm for correcting Byzantine faults. The solution in that paper uses the older version of fused data structures [34]. In this section, we present those algorithms and proofs with minor extensions for the current version of fused data structures.

To correct Byzantine faults, we maintain f additional copies of each primary that enable efficient *detection* of Byzantine faults. This maintains the invariant that there is at least one correct copy in spite of f Byzantine faults. We also maintain f fused backups for the entire set of primaries, which is used to identify and *correct* the Byzantine primaries, after the detection of the faults. Thus, we have a total of $nf + f$ backup data structures. The only requirement on the fused backups $\{F_j, j = 1..f\}$ is that if F_j is not faulty, then given the state of any $n-1$ data structures among $\{X_1 \ldots X_n\}$, we can recover the state of the missing one. Thus, a simple XOR or sum based fused backup is sufficient. Even though we are correcting f faults, the requirement on the fused copy is only for a single fault (because we are also using replication).

The primary X_i and its f copies are called *unfused* copies of X_i. If any of the $f + 1$ unfused copies differ, we call the primary, *mismatched*. Let the state of one of the primaries X_i be v. The number of unfused copies of X_i with state v is called the *multiplicity* of X_i.

```
Client:
    send update to all unfused $f + 1$ copies;
    if (all $f + 1$ responses identical)
        use the response;
    else invoke recovery algorithm;

Unfused Copies:
    on receiving any message from client,
        update local copy;
        send state update to fused processes;
        send response to the client;

Fused Copies:
    on receiving updates from unfused copies,
        if (all $f + 1$ updates identical)
            carry out the update;
        else invoke recovery algorithm;
```

Figure 5.10: Detecting Byzantine Faults [33]

We describe an algorithm in Fig. 5.10, 5.11 and 5.12 to correct f Byzantine faults. We keep f copies for each primary and f fused data structures overall. This results in additional $nf + f$ data structures in the system. If there are no faults among the unfused copies, all $f + 1$ copies will result in the same output and therefore the system will incur the same overhead as the replication-based approach. If the client or one of the fused backups detects a mismatch among the values received from the unfused copies, then the recovery algorithm is invoked. The recovery algorithm first reduces the number of mismatched primaries to one and then uses the locate algorithm to identify the correct primary. We describe the algorithm in greater detail in the following paragraphs.

The recovery algorithm in Fig. 5.11 first checks the number of primaries

```
Recovery Algorithm:
    Accquire all available data structures;
    Let $t$ be the number of mismatched primaries;
    while ($t > 1$) do
        choose a copy of some primary $X_i$ with largest multiplicity;
        restart unfused copies of $X_i$ with the state of the chosen copy;
        $t = t - 1$;
    endwhile;
    // Can assume that $t$ equals one.
    // Let $X_c$ be the mismatched primary.
    Locate faulty copy among unfused copies of $X_c$
    using the locate algorithm;
```

Figure 5.11: Correcting Byzantine Faults: Detect Faulty Primaries [33]

that are mismatched. First consider the case when there is a single mismatched primary, say X_c. Now given the state of all other primaries, we can successively retrieve the state of X_c from fused data structures $F_j, j = 1..f$ until we find a copy of X_c that has $f + 1$ multiplicity.

Lemma 4. Assume that among the n primaries, there is a mismatch for at least two primaries, say X_c and X_d. Let $\alpha(c)$ and $\alpha(d)$ be the largest multiplicity among unfused copies of X_c and X_d respectively. Without loss of generality, assume that $\alpha(c) \geq \alpha(d)$. The copy with multiplicity $\alpha(c)$ is correct.

Proof. If this copy is not correct, then there are at least $\alpha(c)$ liars among unfused copies of X_c. We now claim that there are at least $f + 1 - \alpha(d)$ liars among unfused copies of X_d which gives us the total number of liars as $\alpha(c) + f + 1 - \alpha(d) \geq f + 1$ contradicting the assumption on the maximum number of liars. Consider the copy among unfused copies of X_d with multiplicity $\alpha(d)$. If this copy is correct we have $f + 1 - \alpha(d)$ liars. If this copy is incorrect, we

```
*Locate Algorithm*:
    **Input**: primary $X_c$ with mismatched copies;
    $Z$: set of unfused copies of $X_c$;
    Discard copies in $Z$ and fused backups
    with wrong index/aux structures;
    **while** (there are mismatched copies in $Z$)
        $w = min\{r : \exists p, q \in Z : value_p[r] \neq value_q[r]\}$;
        $Y$: set of values of $state[w]$ for each copy in $Z$;
        $j = 1$;
        **while** (no value in $Y$ with multiplicity $f + 1$)
            create, $v$=$state[w]$ using $F_j$ and all $X_i, i \neq c$;
            add $v$ to $Y$;
            $j = j + 1$;
        **endwhile**;
        delete copies from $Z$ in which $state[w] \neq v$;
    **endwhile**;
```

Figure 5.12: Correcting Byzantine Faults: Correct Faulty Primary [33]

know that the correct value has multiplicity less than or equal to $\alpha(d)$ and therefore there are at least $f + 1 - \alpha(d)$ liars among unfused copies of X_d. Hence, the primary with multiplicity $\alpha(c)$ is correct. □

By identifying the correct primary, we have reduced the number of mismatched primaries by one. By repeating this argument, we get to the case when there is exactly one mismatched primary, say X_c. We use the locate algorithm in Fig. 5.12 to locate the correct copy of X_c.

In the locate algorithm, we first identify errors in the auxiliary and index structures. Since this information is replicated at all the f fused backups, we can obtain $2f + 1$ versions of this information among which at least $f + 1$ versions are identical (at most f liars). The remaining f versions are certainly faulty and unfused copies with this information can be discarded. If there are no errors among the auxiliary/index structures, we identify errors in the data elements.

The set Z in Fig. 5.12 maintains the invariant that it includes all the correct unfused copies (and may include incorrect copies as well). The invariant is initially true because all indices from $1..f+1$ are in Z. Since the set has $f+1$ indices and there are at most f faults, we know that the set Z always contains at least one correct copy.

The outer *while* loop of the locate algorithm, iterates until all copies are identical. If all copies in Z are identical, from the invariant it follows that all of them must be correct and we can simply return any of the copies in Z. Otherwise, there exist at least two different copies in Z, say p and q. Let w be the first key in which states of copies p and q differ. Either copy p or the copy q (or both) are liars. We now use the fused data structures to recreate copies of $state[w]$, the value associated with key w.

Since we have the correct copies of all other primaries $X_i, i \neq c$, we can use them with the fused backups $F_j, j = 1..f$. Note that the fused backups may themselves be wrong so it is necessary to get enough multiplicity for any value to determine if some copy is faulty. Suppose that for some v, we get multiplicity of $f+1$. This implies that any copy with $state[w] \neq v$ must be faulty and therefore can safely be deleted from Z. We are guaranteed to get a value with multiplicity $f+1$ out of total $2f+1$ values, viz. $f+1$ values from unfused copies of X_c and f values decoded using the f fused backups and remaining correct primaries. Further, since copies p and q differ in $state[w]$, we are guaranteed to delete at least one of them in each iteration of the inner while loop. Eventually, the set Z would either be singleton or will contain only identical copies, which implies that we have located a correct copy.

Our approach combines the advantages of replication and coding theory. We have enough replication to guarantee that there is at least one correct copy at all times and therefore we do not need to decode the entire state data structure but only locate the correct copy. We have also taken advantage of coding theory to reduce the number of copies from $2f$ to f. It can be seen that our algorithm is optimal in the number of unfused and fused backups it maintains to guarantee that there is at least one correct unfused copy and that faults of any f data structures can be corrected. The first requirement dictates that there be at least $f + 1$ unfused copies and the recovery from Byzantine faults requires that there be at least $2f + 1$ fused or unfused copies in all.

Time Complexity Analysis: The recovery algorithm has two components. First we reduce the number of mismatched primaries to one. Then, we identify the faulty primary using the locate algorithm. To reduce the mismatched primaries, we use the while loop in Fig. 5.11, which can have at most t iterations. In each iteration we find the copy with the greatest multiplicity among the $f + 1$ unfused copies of each mismatched primary. There can be $O(t)$ mismatched primaries and each primary has $O(m)$ nodes of size $O(s)$ each. Hence, the time complexity of each iteration of the while loop in Fig. 5.11 is $O(mstf)$. So, the time complexity of reducing the mismatched primaries to one is $O(msft^2)$.

We now analyze the time complexity of the *locate* algorithm. Checking for errors in the auxiliary/index structures, which contain m pointers, can be performed in in $O(mf)$ time across all the f fused backups. Assume that there are $t \leq f$ actual faults that occurred. We delete at least one unfused copy of X_c in each iteration of the outer *while* loop and there are at most t faulty data structures giving us the bound of t for the number of iterations of the while loop. In each iteration, creating $state[w]$ requires at most $O(s)$ state to be decoded at each fused data structure at the cost of $O(ns)$. The maximum number of fused data structures that would be required is t. Thus, $O(nts)$ work is required for a single iteration of the outer while loop before a copy is deleted from Z. To determine w in incremental fashion requires $O(mfs)$ work cumulative over all iterations. Combining these costs we get the complexity of the locate algorithm to be $O(mfs + nst^2)$. So overall, the time complexity of the recovery algorithm is $O(nst^2 + msft^2)$.

Recovery in replication reduces to finding the state with $t + 1$ votes among the $2f + 1$ copies of each primary, where t is the actual number of faults. Since this majority can be found by inspecting at most $2t+1$ copies among the primaries, recovery has time complexity $O(mst)$, where m is the number of nodes in each data structure and s is the size of each data structure. In the following section, we summarize the results of this chapter through a comparison of replication with fusion.

5.5 Comparison of Replication and Fusion

In this section, we describe the main differences between replication and fusion (Table 5.1). Throughout this section, we assume n primary data structures, containing at most $O(m)$ nodes of size $O(s)$ each. Each primary can be updated in $O(p)$ time. We assume that the system can correct either f crash faults or f Byzantine faults, and t is the actual number of faults that occur. Note that, the comparison in this section is independent of the type of data structure used. We assume that the fusion operator is RS coding, which only requires f backup blocks to correct f erasures among a given set of data

Table 5.1: Replication vs. Fusion for Data Structures

	Replication Crash	**Fusion Crash**	**Repilication Byzantine**	**Fusion Byzantine**
Number of Backups	nf	f	$2nf$	$nf + f$
Backup Space	$nmsf$	msf	$2nmsf$	$nmsf + msf$
Max Load/Backup	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Normal Operation Time	$O(p)$	$O(p)$	$O(p)$	$O(p)$
Recovery Time	$O(mst)$	$O(mst^2n)$	$O(mst)$	$O(msft^2 + nst^2)$
Normal Operation Messages	f msgs, size s each	f msgs, size $2s$ each	$2f$ msgs, size s each	f msgs size s, f msgs size $2s$
Recovery Messages	t msgs, size ms each	$n+f-t$ msgs, size ms each	$2t + 1$ msgs, size ms each	$nf + n + f$ msgs, size ms each

blocks.

Number of Backups 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.

Backup Space For crash faults, the total space occupied by the fused backups is msf (f backups of size ms each) 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.

Maximum Load on any Backup We define load as the number of primaries each backup has to service. Since each fused backup has to receive requests from all n primaries the maximum load on the fused backup is n times more than the load for replication. Note that, the higher the value of n, more the savings in space/number of backups ($O(n)$ times), but more the maximum load on any backup (again, $O(n)$ times).

Normal (fault-free) Operation Time The fused backups in our system can be updated with the same time complexity as that for updating the corresponding primary i.e., $O(p)$. We have shown that the updates at the backup can be received in any order and hence, there is no need for synchrony. Also, if Byzantine faults/liars need to be detected with every update in a system, then fusion causes no overhead in time.

Recovery Time This parameter refers to the time complexity of recovery at the client, after it has acquired the state of the relevant data structures. In the case of fusion, to recover from t ($t \leq f$) crash faults, we need to decode the backups with total time complexity $O(mst^2n)$. For replication, this is only $O(mst)$. For Byzantine faults, fusion takes $O(mfs + nst^2)$ to correct t Byzantine faults. In the case of replication, on detecting a mismatch among the $2f + 1$ responses, the client needs to obtain the majority among these copies. As there are t liars in the system, replication only needs to look at $2t + 1$ copies, before obtaining a majority. Since each copy contains $O(m)$ nodes of size $O(s)$ each, this takes time complexity $O(mst)$. Thus, replication is much more efficient than fusion in terms of the time taken for recovery. However, for small values of f (for most applications, $f = 1$ or $f = 2$) the cost of recovery may be acceptable.

Normal (fault-free) Operation Messages This parameter refers to the number of messages that the primary needs to send to the backups for any update. We assume that the size of the key for insert or delete is insignificantly small as compared to the data values. In fusion, for crash faults, every update sent to the primary needs to be sent to f backups. The size of each message is $2s$ since we need to send the new value and old value to the backups. For deletes, the size of each message is $2s$ since we need to send the old value and the value of the top-of-stack element (as shown in Fig. 5.5). Hence, for crash faults, in fusion, for any update, f messages of size $2s$ need to be exchanged. For replication, in inserts, only the new value needs to be sent to the f copies of the primary and for deletes, only the key to be deleted needs to be sent. Hence, for crash faults in replication, for any update f messages of size at most s need to be exchanged.

For Byzantine faults, for fusion, since we maintain f copies of each primary and f fused backups, it needs f messages of size s and f messages of size $2s$ respectively. In replication, $2f$ messages of size s need to be sent to the $2f$ copies of the primary for inserts and for deletes, only $2f$ keys need to be sent.

Recovery Messages 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 $n + f - t$ 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 $nf + f + f$ 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.

Chapter 6

Practical Evaluation

In this chapter, we consider the practical use of our fusion-based solutions. For fused state machines, we present a design for the *grep* application in the MapReduce framework [24] that requires only 1.4 million worker tasks as compare to the 1.8 million worker tasks required by replication. We provide a design tool to generate fused state machines, based on our fusion algorithm. Our evaluation of this tool on the commonly used MCNC'91 benchmarks [95] shows that fusion achieves 38% savings in state space over replication.

For fused data structures, we present a design for fault tolerance in Amazon's Dynamo key-value store [25] that requires just 120 backups as compared to the 300 backups required by replication. Further, we present a library/package of fused data structures for all the containers in the Java Collection framework. Our experimental evaluation of fused data structures in a distributed implementation confirms that fusion is $O(n)$ times more space efficient as compared to replication, where n is the number of primaries.

In our design for Dynamo and MapReduce, we use a combination of fusion and replication, as compared to a pure replication-based solution. While fusion achieves the savings in space, the partial replication minimizes the overhead for normal operation. In the following section, we address the practical challenges faced in building systems that combine fusion and replication.

6.1 Building Fusion-based Fault Tolerant Systems

In a pure replication-based system there will be no fused backups, while in a pure fusion-based system there will be no replicas. However, for most real-world systems, in which fault tolerance and load-balancing are both serious concerns, we need to consider a hybrid solution with both fusion and replication. This section is based on the standard tutorial for the replicated state machine approach [83].

We maintain the following types of processes in our system:

- *Primaries and their copies*: We are given a set of n distinct primary state machines $P_1 \ldots P_n$, each of which model the computation of the system. For each primary, we maintain a set of identical copies for both fault tolerance and load-balancing. The primary P_i with its copies is referred to as the *ensemble* of P_i. For convenience, we refer to any of the machines in the ensemble of P_i as primary P_i.
- *Fused Backups*: These are the backup machines used purely for fault tolerance. We describe the design of these fused backups in detail in chapters 4 and 5. However, for the purpose of overall system design, we note just the following two facts about fused backups: (i) the fused backups need to receive and act on every distinct request addressed to each P_i, for $i : 1$ to n and, (ii) to recover the state of any failed process using the fused backups, we need the state of one primary from each primary ensemble and the state of all available fused backups.

In our system, the fault-free clients of the state machines send requests addressed to the primaries. Based on how these requests are propagated to the copies of the primaries and the fused backups, we defined two models:

- *Client-centric* model: In this model, a client sends the request addressed to a certain primary P_i to all the primaries in the ensemble of P_i and all the fused backups.

- *Primary-centric* model: In this model, a client sends the request addressed to a certain primary P_i to one of the primaries in the ensemble of P_i. This designated primary sends the requests to other primaries in the ensemble of P_i and all the fused backups.

The choice of the appropriate model among these two will be dictated by the system under consideration. For example, a system that uses fused data structures will have to be based on the primary-centric model, since the fused data structures require information from the primaries. On the other hand, the client-centric model maybe more suited for a system with fused state machines. Detection and correction of faults is performed by any of the clients. The key requirements for building a fault tolerant system are as follows:

- C1 (*Agreement*): Requests addressed to a primary P_i must be received by all the primaries in the ensemble of P_i and all the fused backups.

- C2 (*Order*): Requests addressed to a primary P_i must be processed by all the primaries in the ensemble of P_i and all the fused backups in the same relative order.

- C3 (*Commutativity*): The state of the fused backups is independent of the order in which they process requests addressed to different primaries.

In chapters 4 and 5, we prove that our design achieves commutativity. In the following sections, we focus on techniques to guarantee agreement and order in the presence of both crash and Byzantine faults.

6.1.1 Agreement

In the client-centric model, the non-faulty client sends the requests to the primary ensemble and the fused backups. Since processes cannot change their identity, each of the processes in the system accepts requests only from the client. So even if there are Byzantine faults among the primaries or the fused backups, they cannot send erroneous requests. Hence, trivially, the agreement condition C1 is achieved.

In the primary-centric model, it is more difficult to achieve agreement. In this model, the processes receive requests either directly from the client or from other processes in the system. Hence, a designated process may either not send a client request (due to a crash fault) or in the worst case, a non-designated Byzantine process might send an erroneous request to break the system. So, for the primary-centric model, we need to follow a consensus or agreement protocol in which the client is modeled as one of the processes.

Consensus or agreement in the presence of faults is a well studied problem in distributed systems [17, 19, 20, 30, 47, 52] and it guarantees the following properties: (*i*) all non-faulty processes agree on the same value and, (*ii*) if the transmitter of the value (in this case, the client) is non-faulty, then all non-faulty processes use this as the agreed upon value. Hence, based on this protocol, we can achieve agreement even in the primary-centric model. In the following section, we present a technique for achieving condition C2, i.e., order.

6.1.2 Order

Since there are multiple clients sending requests addressed to the same primary, in both the client-centric and primary-centric model, we need to ensure that the primary ensemble and the fused backups process the requests in the same order. The first step is to define a total ordering mechanism that assigns unique identifiers (referred to as *time-stamps*) to each request. Let the time-stamp assigned to request r be denoted by $T(r)$. The ordering mechanism must satisfy these two conditions:

- If r_1 and r_2 are two requests made by a client to a certain primary P_i such that r_1 was made before r_2, then $T(r_1) < T(r_2)$.
- If a request r_1 made by a client to a primary P_i causes another request r_2 to be made by some other client to P_i, then $T(r_1) < T(r_2)$.

We can use any of the total ordering mechanisms in the literature [26, 63] that achieves these two conditions, such as Lamport's logical clocks [50] or synchronized real-time clocks [22, 78].

Each process hosting either a primary or backup has a queue of requests among which it needs to choose a request to act on next. To achieve the order condition C2, each process in the system must next act on a request r that satisfies the following conditions:

1. Request r must have the lowest unique time-stamp $T(r)$ among all requests in the queue.
2. The process must have received at least one request from each client with time stamp greater than $T(r)$.

Ties among the requests are broken arbitrarily. To implement the second condition mentioned above, each client can periodically send "null" requests to each process in the system. In the following section we focus on the recovery of the failed processes.

6.1.3 Recovery

When a client of the system detects a fault among either the primary ensemble or the fused backups, it acquires the state of all the necessary processes and performs recovery to regenerate the state of the failed processes. The actual details of the recovery operation are specific to the nature of the backups and this is explained in chapters 4 and 5. In this section, we focus on the steps to ensure the correctness of the recovery operation.

As mentioned in section 1.3, the clients of the processes receive the outputs of the processes and can hence detect faults among them. When faults are detected, say among the primary ensemble and the fused backups, these are the sequential steps the client needs to follow to ensure correct recovery of each failed process.

Step 1, Recover failed primary processes in whose ensemble at least one copy remains: For each such process P_i, the client acquires the state and request queues of any one of the primaries in the ensemble of P_i. Then, it restarts the failed primary processes in this ensemble with the state and request queue of this selected primary.

Let the request with the greatest time-stamp in this queue be denoted by r. Let r_c be the first request received by the restarted process. The client relays all the requests whose time-stamps are greater than or equal to r and lesser than or equal to r_c to the restarted process. Now the restarted process can start executing.

Step 2, Recover remaining failed processes: The client acquires the state and request queues of all the fused backups and the state and request queue of one primary (if available) from the ensembles of all the primaries. Let the processes from which the client acquired the state and request queues be referred to as the *recovery processes.* The recovery processes on sending their state and request queues to the client, stop acting on any client requests.

Among all the acquired request queues, let the request with the largest time-stamp addressed to primary P_i be denoted by r_i. The client first ensures that each distinct primary P_i among the recovery processes executes up to request r_i. The client then ensures that all the fused backups among the recovery processes execute up to r_i for each value of i. The client can now re-acquire the state and request queue of the recovery processes and recover the state of the failed processes. The recovery processes can now resume execution.

The client restarts the failed processes with the recovered state. Let r_c be the first request received by the restarted process. If the restarted process is a primary P_i, the client relays all the requests whose time-stamps are greater than or equal to r_i and lesser than or equal to r_c to the restarted process. Let the request with the smallest time-stamp among r_i, for each value of i, be denoted by r_m. If the restarted process is a fused backup, then the client has to relay all the requests with time-stamps greater than r_m and lesser than or equal to r_c. Now the restarted processes can start executing.

In this section, we have seen the various aspects of system design while building systems that combine fusion with replication. In the following section, we outline our solution for the MapReduce framework.

6.2 Fusion-based Grep in MapReduce

In many large scale distributed stream processing applications, active replication is often used for fault tolerance [8, 84]. A common function on the streams, is the *grep* function, which checks if every line of the file matches patterns defined by regular expressions (modeled as DFSMs). These distributed applications can be modeled using the MapReduce framework [24]. Typically, the Map-Reduce framework is built using the master-worker configuration where the master assigns the map and reduce tasks to various workers. While the map tasks perform the actual computation on the data files received by it as <key, value> pairs, the reducer tasks aggregate the results according to the keys and writes them to the output file.

In this section, we compare the existing replication-based design of the grep application in the MapReduce framework with a hybrid fusion-based design (Fig. 6.1). Specifically, we assume that the regular expressions for grep are $((0+1)(0+1))*$, $((0+2)(0+2))*$ and $(00)*$ modeled by A, B, C shown in Fig. 4.1. We show that the current replication based solution requires 1.8 million map tasks while our solution that combines fusion with replication requires only 1.4 million map tasks. This results in considerable savings in space and other computational resources.

6.2.1 Existing MapReduce Design

We first outline a simplified version of a pure replication based solution to correct two crash faults. Given an input file stream, the master splits the file into smaller partitions (or streams) and breaks these partitions into <file name, file content> tuples. For each partition, we maintain three primary map tasks m_A, m_B and m_C that output the lines that match the regular expressions

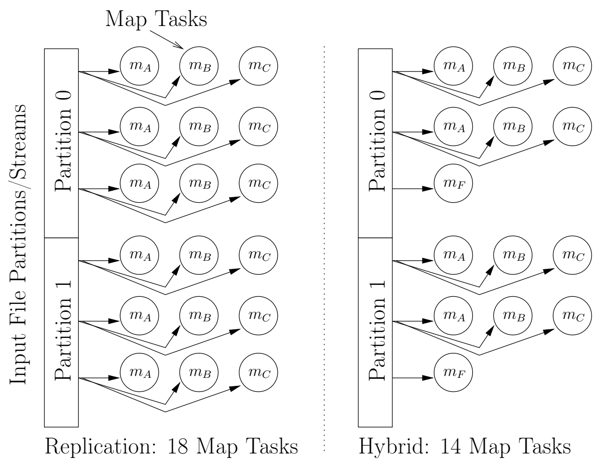


Figure 6.1: Replication vs. Fusion for *grep* in MapReduce.

modeled by A, B and C respectively. To correct two crash faults, we maintain two additional copies of each primary map task for every partition. The master sends tuples belonging to each partition to the primaries and the copies. The reduce phase just collects all lines from these map task and passes them to the user. Note that, the reducer receives inputs from the primaries and its copies and simply discards duplicate inputs. Hence, the copies help in both fault tolerance and load-balancing.

When map tasks fail, the state of the failed tasks can be recovered from one of the remaining copies. From Fig. 6.1, in replication, it is clear that each file partition requires nine map tasks. In such systems, typically, the input files are large enough to be partitioned into 200,000 partitions [24]. Hence, replication requires 1.8 million map tasks.

6.2.2 Hybrid Fusion-based Design

In this section, we outline an alternate solution based on a combination of replication and fusion. For each partition, we maintain just one additional copy of each primary and also maintain one fused map task, denoted m_F for the entire set of primaries. The fused map task searches for the regular expression (11)* modeled by F_1 in Fig. 4.1. Clearly, this solution can correct two crash faults among the primary map tasks, identical to the replication-based solution. The reducer operation remains identical. The output of the fused map task is relevant only for fault tolerance and hence it does not send its output to the reducer. Note that since there is only one additional copy of each primary, we compromise on the load balancing as compared to pure replication. However, we require only seven map tasks as compared to the nine map tasks required by pure replication.

When only one fault occurs among the map tasks, the state of the failed map task can be recovered from the remaining copy with very little overhead. Similarly, if two faults occur *across* the primary map tasks, i.e., m_A and m_B fail, then their state can be recovered from the remaining copies. Only in the relatively rare event that two faults occur among the copies of the same primary, does the fused map task have to be used for recovery. For example, if both copies of m_A fail, then m_F needs to acquire the state of m_B and m_C (any of the copies) and perform the algorithm for crash correction in 4.4.2.1 to recover the state of m_A.

Considering 200,000 partitions, the hybrid approach needs only 1.4 million map tasks which is 22% less than the map tasks required by replication. Note that as n increases, the savings in the number of map tasks increases even further. This results in considerable savings in terms of (*i*) the state space required by these map tasks and, (*ii*) resources such as the power consumed by them. In the following section, we describe the implementation of our design tool to generate efficient backups, followed by the experimental evaluation.

6.3 Fused DFSM Design Tool

In this section, we evaluate our DFSM design tool [4], implemented in Java, based on the incremental version of the *genFusion* algorithm presented in section 4.2.3. The input to the tool is a set of n primary machines $\mathcal{P}$, the number of faults f that need to be corrected, the state reduction $\triangle s$ and the event reduction $\triangle e$ required. The output of the tool is a minimal set of f backups that can correct f crash or $\lfloor f/2 \rfloor$ Byzantine faults among $\mathcal{P}$. Each backup contains at most $N - \triangle s$ states and $|\Sigma| - \triangle e$ events, where N is the number of states in the reachable cross product of the primaries and Σ is the union of the events sets of the primaries.

The parameters $\triangle s$ and $\triangle e$ can be chosen according to the requirements of the specific application in concern. For example, if the amount of memory allocated for each backup is 10 bits, then the number of states in each backup cannot exceed $2^{10} = 1024$ states. So, the user can first determine the number of reachable cross product states, N, and then specify $\triangle s = max(N - 1024, 0)$. Similarly, the bandwidth allocated for each backup could be 10Mbps. Assuming that the bandwidth required for each input/event stream is 1Mbps and the union of events among the primaries is Σ, the user can specify $\triangle e = max(|\Sigma| - 10, 0)$. Note that, this is just one potential way to generate $\triangle s$ and $\triangle e$. Different systems may have different constraints that need to be translated to these parameters.

6.3.1 Experiments and Results

In this section, we evaluate our tool using the MCNC'91 benchmarks [95] for DFSMs, widely used for research in the fields of logic synthesis and finite state machine synthesis [64, 96]. We compared the performance of fusion with replication for 100 different combinations of the benchmark machines, with $n = 3$, $f = 2$, $\triangle e = 3$ and present some of the results in Table 6.1. The implementation with detailed results are available in [4].

Let the primaries be denoted P_1, P_2 and P_3 and the fused-backups F_1 and F_2. Column 1 of Table 6.1 specifies the names of three primary DFSMs. Column 2 specifies the backup space required for replication ($\prod_{i=1}^{1=3} |P_i|^f$), column 3 specifies the backup space for fusion ($\prod_{i=1}^{i=2} |F_i|$) and column 4 specifies the percentage state space savings ((column 2-column 3)* 100/column 2). Column 5 specifies $|\Sigma|$, column 6 specifies the average number of events across F_1 and F_2 and the last column specifies the percentage reduction in events ((column 5-column 6)*100/column 5).

The average state space savings in fusion (over replication) is 38% (range 0-99%) over the 100 combination of benchmark machines, while the average event-reduction is 4% (range 0-45%). We also present results in [4] that show that the average savings in time by the incremental approach for generating the fusions (over the non-incremental approach) is 8%. Hence, fusion achieves significant savings in space for standard benchmarks, while the event-reduction indicates that for many cases, the backups will not contain a large number of events. In the following section, we present the practical evaluation of our fused data structures.

6.4 Fusion-based Key-Value Store

In this section, we illustrate the usefulness of fused data structures in a real-world distributed system. Amazon's Dynamo [25] 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-

Table 6.1: Evaluation of Fusion on the MCNC'91 Benchmarks

Machines	*Replication State Space*	*Fusion State Space*	*% State Savings*	$\lvert\Sigma\rvert$	*Fusion Events*	*% Event Reduction*
dk15, bbara, mc	25600	19600	23.44	16	10	37.5
lion, bbtas, mc	9216	8464	8.16	8	7	12.5
lion, tav, modulo12	36864	9216	75	16	16	0
lion, bbara, mc	25600	25600	0	16	9	43.75
tav, beecount, lion	12544	10816	13.78	16	16	0
mc, bbtas, shiftreg	36864	26896	27.04	8	7	12.5
tav, bbara, mc	25600	25600	0	16	16	0
dk15, modulo12, mc	36864	28224	23.44	8	8	0
modulo12, lion, mc	36864	36864	0	8	7	12.5

based solution which is simple to maintain but expensive in terms of space. We propose an alternate design using a combination of both fusion and replication, which consumes far less space, while guaranteeing nearly the same levels of durability and availability.

6.4.1 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 additional copies of each primary hash table are maintained. 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 copies that must participate in each read request and write request respectively. These values are each chosen to be less than f. In Fig. 6.2 (i), we illustrate a simple set up of Dynamo for $n = 4$ primaries, with $(f, w, r) = (3, 2, 2)$.

To read and write from the data store, the client can send its request to any one of the $f+1$ copies responsible for the key of the request, and designate it as 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 backup copies 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 copies 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 data structures crash, the remaining copies responsible for the same key space can take over all requests addressed to the failed data structures. Once the crashed data structure comes back, the copy that was acting as proxy just transfers back the keys that were meant for the node. In Fig. 6.2 (i), since there can be at most three crash faults in the system, there is at least one node copy for each primary remaining for recovery.

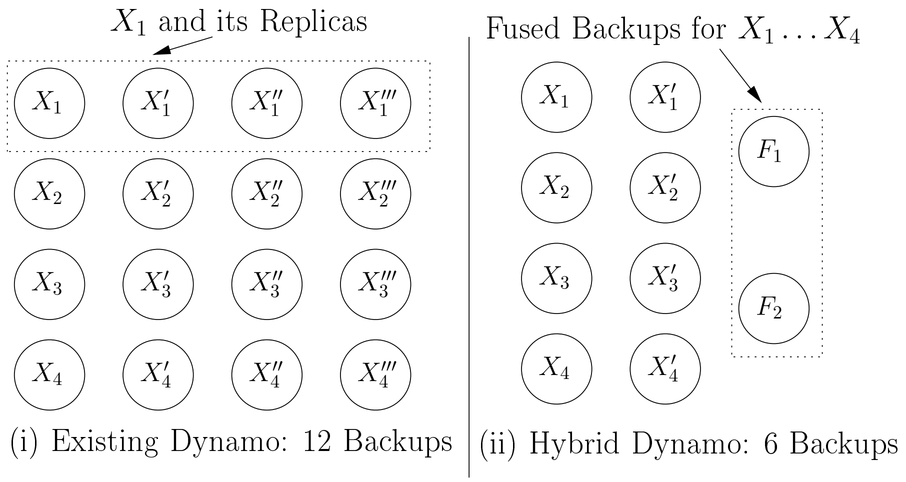

X_1 and its Replicas
X_1 X_1' X_1'' X_1'''
X_2 X_2' X_2'' X_2'''
X_3 X_3' X_3'' X_3'''
X_4 X_4' X_4'' X_4'''
(i) Existing Dynamo: 12 Backups
Fused Backups for $X_1 \ldots X_4$
X_1 X_1'
X_2 X_2'
X_3 X_3'
X_4 X_4'
F_1
F_2
(ii) Hybrid Dynamo: 6 Backups

Figure 6.2: Design Strategies for Dynamo

6.4.2 Hybrid Fusion-based Design

We propose a hybrid design for Dynamo that uses a combination of fusion and replication. We focus on the case of $(f, w, r) = (3, 2, 2)$. Instead of maintaining three additional copies for each primary ($f = 3$), we maintain just a single additional copy for each primary and two fused backups for the entire set of primaries as shown in Fig. 6.2 (*ii*). The fused backups achieve the savings in space while the additional copies allow the necessary availability for reads. The fused backups along with the additional copies 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 backups cannot directly respond to the client requests since they require the old value associated with the key (section 5.2). 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 case of $w = 2$, as long as the coordinator, say X_i obtains a response from one among the three backups (one copy and two fused backups) the write can succeed. This is similar to the existing design and hence performance for writes is not affected significantly. On the other hand, performance for reads does drop since the fused backups that contain data in the coded form cannot return the data value corresponding to a key in an efficient manner. Hence, the two additional copies 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, trading some of the space efficiency for availability. For the set up shown in Fig. 6.2, we can maintain four fused backups where F_1, F_2 are the fused backups for X_1 and X_2, while F_3 and F_4 are the fused backups of X_3 and X_4.

Similar to the existing design of Dynamo, when data structures crash, if there are surviving copies responsible for the same keys, then they can take over operation. However, since we maintain only one additional copy per primary, it is possible that none of the copies 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 copies 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 each containing a primary. So in a typical cluster with $n = 100$, $f = 3$ the original approach requires, $n * f = 300$ backup data structures. Consider a hybrid solution that maintains one additional copy for each primary and maintains two fused backups for every 10 primaries. This approach requires only $100 + 20 = 120$ backup data structures. This results in savings in space, as well as power and other resources required by the processes running these data structures. Hence, the hybrid solution can be very beneficial for such a real-world system. In the following section, we experimentally evaluate our design of fused data structures.

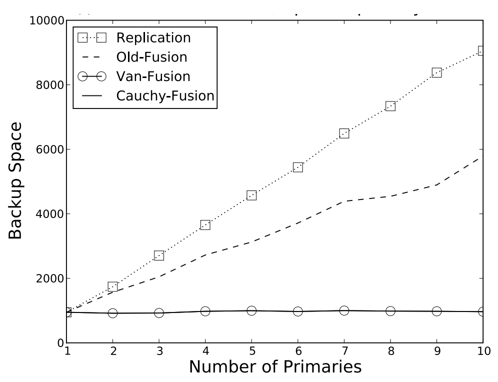
6.5 Fused Data Structure Library

In this section, we describe our fusion-based data structure library [2] that includes all data structures provided by the Java Collection Framework. Further we have implemented our fused backups using Cauchy RS codes (referred to as *Cauchy-Fusion*) and Vandermonde RS codes (*Van-Fusion*). We refer to either of these implementations as the *current* version of fusion. We have compared its performance against replication and the older version of fusion (*Old-Fusion*) [34]. Old-Fusion has a different, simpler design of the fused backups, similar to the one presented in the design motivation of section 5.2. We extend it for f-fault tolerance using Vandermonde RS codes. The current versions of fusion, using either Cauchy or Vandermonde RS, outperform 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 achieve almost n times savings in space as confirmed by the theoretical results, while not causing too much update overhead. Recovery is much cheaper in replication.

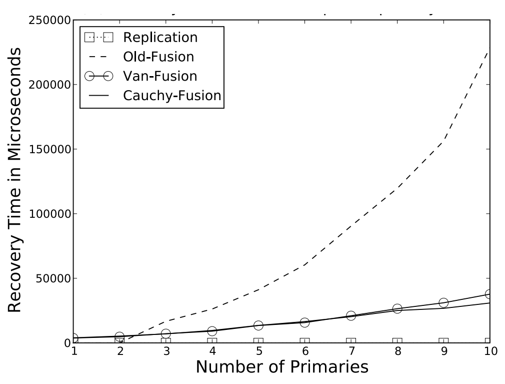
6.5.1 Experiments and Results

We implemented fused backups and primary wrappers for the data structures in the Java Collection framework that are broadly divided into list-based, map-based, set-based and queue-based data structures. We evaluated the performance of a representative data structure in two of these categories: linked lists for list-based and tree maps for map-based data structures. Both Old-Fusion and Van-Fusion use Vandermonde RS codes with field size 2^{16}, while Cauchy-Fusion uses Cauchy RS codes, with field size 2^5 (refer to section 3.2 for details of coding theory). The RS codes we have used are based on the C++ library provided by James S. Plank [74, 75]. Currently we just support the Integer data type for the data elements at the primaries.

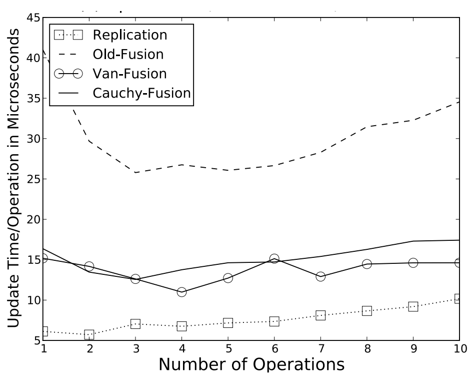
We implemented a distributed system of hosts, each running either a primary or a backup data structure and compared the performance of the four solutions: Replication, Old-Fusion, Van-Fusion and Cauchy-Fusion. The algorithms were implemented in Java with TCP sockets for communication and the experiments were executed on a single Intel quad-core PC with 2.66 GHz clock frequency and 12 GB RAM. In the future, we wish to evaluate fusion over physically disparate machines. 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 five runs. In our experiments, we only assume crash faults. We describe the results for the three main tests that we performed for linked lists: backup space, update time at the backup and recovery time (Fig. 6.3). In Fig. 6.4, we present the results of our experiments for tree maps, which is similar to the results for linked lists.



(a) Backup Nodes

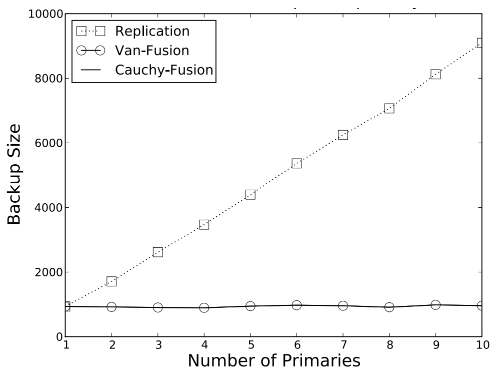


(b) Recovery Time

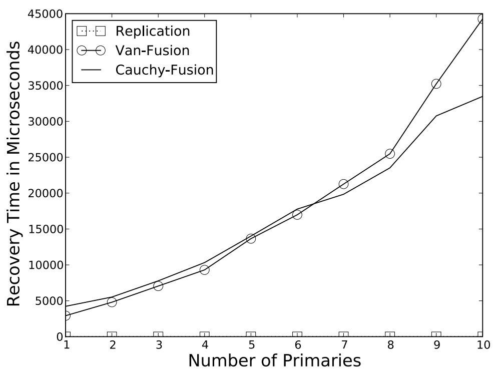


(c) Update Time

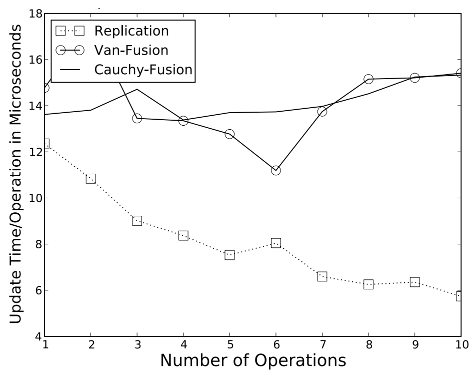
Figure 6.3: Linked Lists: Experimental evaluation.



(a) Backup Nodes



(b) Recovery Time



(c) Update Time

Figure 6.4: Maps: Experimental evaluation.

Backup Nodes To measure the space required by the backups, 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 required by each solution. We fix $f = 3$, $ops = 500$ and vary n from 1 to 10. Cauchy-Fusion and Van-Fusion, differ only in the type of RS code used, but use the same design for the backups. So, they both require the same number of backup nodes. Both Cauchy-Fusion and Van-Fusion perform much better than both replication (approximately n times) and Old-Fusion (approximately $n/2$ times) because the number of nodes per backup never exceeds the maximum among the primaries.

Recovery Time We measure recovery time as the time taken to recover the state of the crashed data structures *after* the client obtains the state of the requisite data structures. The same experiment as that used to measure backup space was used to compare the four solutions. Cauchy-Fusion and Van-Fusion perform much better than Old-Fusion (approximately $n/2$ times) because recovery in fusion involves iterating through all the nodes of each fused backup. The current design contains fewer nodes and hence performs better. The time taken for recovery by replication is negligible as compared to fusion-based solutions (the curve is almost merged with the x-axis in the graphs). This is to be expected since recovery in replication requires just copying the failed data structures after obtaining them. However, note that, even for $n = 10$, the time taken for recovery by both Cauchy and Van-Fusion is under 40 millisecs. This can be a small cost to pay for the considerable savings that we achieve in space.

Further analysis of the recovery times in both Cauchy-Fusion and Van-Fusion shows that almost 40 % of the cost of recovery is spent in decoding the coded data elements. This implies two things. First, using a different code such as LDPC codes, that offers faster decoding in exchange for less space efficiency, fusion can achieve faster recovery times. Second, more than 50 % of recovery time is spent on just iterating through the backup nodes, to retrieve the data for decoding. Hence, optimizing the recovery algorithm, can reduce the recovery time. The other observation is that, even though Cauchy RS codes have much faster decode times than Vandermonde RS codes, the recovery time for Cauchy-Fusion is only marginally better than Van-Fusion. We believe this is mainly due to the small data size (4 byte integers). For larger data values, Cauchy-Fusion might perform much better than Van-Fusion. These are future areas of research that we wish to explore.

Update Time Finally, to measure the update time at the backups, we fixed $n = 3$, $f = 1$ and varied *ops* from 500 to 5000. Both Cauchy-Fusion and Van-Fusion have more update overhead as compared to replication (approximately 1.5 times slower) while they perform better than the older version (approximately 2.5 times faster). Since the current design of fused backups has fewer backup nodes, it takes less time to iterate through the nodes for an update. The update time at a backup can be divided into two parts: the time taken to locate the node to update plus the time taken to update the node's code value. The code update time was insignificantly low and almost all the update time was spent in locating the node. Hence, optimizing the update algorithm can reduce the total update time considerably. This also explains why Cauchy-Fusion does not achieve any improvement over Van-Fusion and at times does slightly worse, because the overhead of dealing with blocks of data in Cauchy-Fusion exceeds the savings achieved by faster updates. As mentioned before, we believe that with the larger data sizes, Cauchy-Fusion may perform as expected.

Chapter 7

Conclusion and Future Work

In this dissertation, we present a fusion-based technique for fault tolerance in distributed systems, in which we maintain far fewer backups than the current replication-based solutions. This leads to two major advantages over replication. First, the total space or memory required for fault tolerance is reduced. Second, we save on the computational resources such as the power required to run the backups. To make our techniques generally applicable, we describe fusion in two separate contexts: (*i*) distributed systems modeled as finite state machines and, (*ii*) distributed systems hosting large data structures. In the following section, we summarize our fusion-based solutions for state machines and data structures.

7.1 Fused State Machines

Given a set of n different deterministic finite state machines (referred to as machines), we correct f crash faults (or $\lfloor f/2 \rfloor$ Byzantine faults) among them using just f additional *fused* machines as compared to the nf backups required by replication. We present a framework for fault tolerance in machines and provide a polynomial time algorithm to generate fused backup machines. Our algorithm ensures that the backups are efficient in terms of the number of states in their state set and the number of events in their event set. Further, we present algorithms to detect and correct faults that incur very little additional overhead as compared to replication.

We provide a Java design tool to generate fused state machines and our experimental evaluation shows that, on average, fusion achieves 38% savings in state space over replication. To illustrate the practicality of our solution, we present a fusion-based design for *grep* in the MapReduce framework that requires 22% fewer worker tasks than a pure replication-based solution.

In our fusion-based solution for state machines, there are two major disadvantages over replication. First, the time complexity of generating the fused backups is exponential in n, while in the case of replication it is linear in n. However, these backups have to be generated only once in the life time of the system. Second, in the worst case, the event set of the fused backup is the union of the event sets of all primaries. This could lead to excessive load on the backups. In the future, we wish to explore techniques to reduce this load.

7.2 Fused Data Structures

Given a set of n different data structures, we correct f crash faults among them using just f additional *fused* data structures as compared to the nf backups required by replication. We show that our solution achieves $O(n)$ savings in space over replication while ensuring that the overhead for normal operation is only as much as the overhead for replication. Further, we present a solution to correct f Byzantine faults using just $nf + f$ backups as compared to the $2nf$ backups required by replication. We present a generic design of fused backups for most commonly used data structures such as stacks, vectors, binary search trees, tree maps and hash tables.

We provide a Java library of these backups and our experimental results show that fusion is space efficient as compared to replication (almost n times), while causing very little overhead for normal operation. To illustrate the applicability of fused data structures, we present a fusion-based design for Amazon's Dynamo key-value store that requires 60% fewer backups than a pure replication-based solution.

In our solution for fused data structures, we use the Reed Solomon [79] erasure codes that guarantee space optimality. However, this leads to two disadvantages over replication. First, the time complexity of recovery is far more than that for replication. Second, each fused backup has to receive events corresponding to all the primaries. We wish to explore other erasure codes that offer different compromises. In the following section, we explore the future avenues of research in fusion.

7.3 Future Work

Building Fusion-based Distributed Systems In this dissertation, we present the fusion-based designs of the Dynamo key-value store and the grep application in the MapReduce framework. We wish to implement these designs and compare their performance with a replication-based solution for various system parameters such as the number of calls to main memory, the end-to-end time for operation, the peak and average loads on each process and the power consumed by the nodes.

Fusion can be used as a framework for fault tolerance in many real-world systems such as peer-to-peer networks [87, 97], streaming computations [8, 84] and distributed file systems [14]. Each of these applications may offer challenges that are widely different from the other. For example, while peer-to-peer networks need to support huge amounts of node churn, streaming applications need to satisfy strict deadlines, despite failures. The design and implementation of fusion-based solutions for these applications is an area of future research.

In [33], the author has developed fusion-based fault-tolerant versions of the vector clock algorithm, a causal ordering algorithm and a mutual exclusion algorithm. This may be extended to many other standard distributed applications such as the total ordering algorithms or arrow protocols.

Backups Outside the Closed Partition Set So far in this dissertation, we have only considered machines that belong to the closed partition set. In other words, given a set of primaries $\mathcal{P}$, our search for backup machines was restricted to those that are less than the RCP of $\mathcal{P}$. It is possible that efficient backups exist outside these set of machines. An interesting avenue of research is to understand the theory of such machines and design efficient algorithms to generate them.

Communicating State Machines for Efficient Backups In our solution for fused state machines, we assume that the primaries and the backups do not communicate with each other. On the other end of the spectrum, consider a solution in which we maintain parity or checksum servers that contain the erasure code corresponding to the states of the primaries. In this solution each primary has to communicate with the backup servers after every event/update. As seen in section 2, such a solution involves considerable overhead for communication among the machines and recovery. However, unlike our fusion-based solution, this simple checksum-based approach guarantees space optimality.

We wish to explore a solution in which we allow partial communication among the state machines to generate efficient backups, while ensuring reasonably less overhead for normal operation and recovery.

Further, in this dissertation, we reduce the RCP of the primaries, to generate machines that act independently of each other, i.e., a *parallel decomposition* of the RCP. In [55], the authors explore the notion of the *serial decomposition* of a given state machine. For example, a machine M maybe decomposed into two machines M_a and M_b such that the input to M_b is dependent on the output of M_a. Given the state of M_a and M_b, if we can uniquely determine the state of M, then $\{M_a, M_b\}$ is a serial decomposition of M. Communicating backups can lead to far more efficient solutions for fusion.

Load-Balancing at the Fused Backups One of the major challenges faced by fusion is the increased load at the fused-backups. If a single process acts as backup for n processes, then the load at the backup process may be quite high for large values of n. There are many ways we propose to deal with additional load at the backup process. First, the objective of primary processes and backup processes may be different. Primary processes may be optimized for fast read-only operations. Backup processes that are used for fault-tolerance do not execute read-only operations and can be optimized for write operations. For example, a set data structure may be implemented as a red-black tree for primary use, whereas, the backup may be organized as a simple linked list.

Second, in many distributed computing examples, the operation at the fused process can be aggregated, thereby reducing the total number of operations performed at the backup. The amount of work performed at the fused process is crucially dependent on the fusion algorithm used. For example, if an update is common to all the primaries, then there is no reason for each primary to send this update to the fused process. Just one aggregated update corresponding to this common update can be applied on the fused process. Finally, fusion can coexist with replication. So instead of one fused process for n primary processes, we can have a system with m fused processes each of them acting as backup for n/m primary processes. We will investigate efficient mechanisms that change the degree of replication to adapt to different workloads.

Quantifying the Trade-off between Space and Load In recent times, there has been extensive work on erasure codes for distributed storage [27, 56, 67, 93]. In particular, there has been considerable work on understanding the trade-off between the amount of data that needs to be stored in each storage node vs. the amount of bandwidth needed for recovery from faults (the storage-bandwidth problem). Further, it has been shown that codes exist for each point on the storage-bandwidth curve. For many systems, where the nodes are exposed directly to client operations, it is important to understand the trade-off between the load on each backup node during normal operation versus the total amount of redundant storage per backup node.

To quantify the load on a backup node/process, we focus on two parameters: (i) *fusion-count*: the number of primaries each process has to service and, (ii) *fusion-bandwidth*: the input bandwidth required by each process. Exploring the trade-offs between the redundant storage per backup node versus the fusion count and the fusion-bandwidth is an interesting area of research. Further, codes could be designed, appropriate to system requirements. For example, replication (which can be considered a special case of fusion) is clearly at one end of the redundancy versus fusion-count curve, since it requires minimum fusion-count but maximum redundancy. Reed-Solomon codes on the other hand are at the other extreme, requiring maximum fusion-count but minimum redundancy.

Alternative Methods for Systematic Coding Currently we use the Reed-Solomon erasure codes for fault tolerance in fused data structures. While these codes guarantee space efficiency, they are inefficient in terms of recovery time complexity. We wish to explore other erasure codes such as Regenerative codes [27, 69], LDPC codes [31, 80, 90] and LT codes [15, 58–60] that offer different trade-offs between various system parameters.

For example, we can use the code presented in [69], to first partition the primary data structures, apply Reed-Solomon codes for each block of these primaries and then apply a simple XOR-based code on these encoded blocks. Such a simple construction ensures that at least single failures can be corrected very efficiently using the XOR encodings. Only if more than one failure occurs (which is the rare case), do we need to use the expensive Reed-Solomon routines for decoding. Hence, overall, at the cost of some space efficiency, we can achieve much better recovery times.

Complex Fault Models In our current fault model, we are mainly concerned with deterministic worst-case failures. We wish to extend this to more complex models such as *stochastic* or *correlated* failures. In the field of information theory, most codes are designed to transmit messages across a noisy channel in which messages can be dropped or corrupted with a certain probability. For example, LDPC or Turbo Codes achieve near optimal rates for message transmission given a certain noise level. It is possible that servers in a distributed system can be modeled as channels with error probabilities. Designing appropriate codes for such a model is an avenue of future research. Also, in this dissertation, we have assumed that failures among the servers are completely independent. In real-world scenarios, failures among the servers are usually correlated [37, 92]. Given a certain failure pattern, it is possible that we can design even more optimized codes for fault tolerance.

Bibliography

[1] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.

[2] 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.

[3] Bharath Balasubramanian and Vijay K. Garg. Fused data structures for handling multiple faults in distributed systems. In *International Conference on Distributed Computing Systems, ICDCS 2011, Minneapolis, Minnesota, USA, June 20-24, 2011*, pages 677–688, 2011.

[4] Bharath Balasubramanian and Vijay K. Garg. Fused fsm design tool (implemented in Java 1.6). In *Parallel and Distributed Systems Laboratory, http://maple.ece.utexas.edu*, 2011.

[5] 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.

[6] Bharath Balasubramanian and Vijay K. Garg. Fault tolerance in distributed systems using fused data structures. *IEEE Transactions on Parallel and Distributed Systems*, 2011 (to appear).

[7] Bharath Balasubramanian, Vinit Ogale, and Vijay K. Garg. Fault tolerance in finite state machines using fusion. In *Proceedings of International Conference on Distributed Computing and Networking (ICDCN) 2008, Kolkata*, volume 4904 of *Lecture Notes in Computer Science*, pages 124–134. Springer, 2008.

[8] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Mike Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *ACM SIGMOD Conf.*, Baltimore, MD, June 2005.

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

[10] Garrett Birkhoff. Lattice theory. volume 25, pages 420 pp+. American Mathematical Society, 1967.

[11] Johannes Blömer, Malik Kalfane, Marek Karpinski, Richard Karp, Michael Luby, and David Zuckerman. An XOR-Based Erasure-Resilient Coding Scheme. Technical Report TR-95-048, International Computer Science Institute, August 1995.

[12] T. L. Booth. *Sequential machines and automata theory.* John Wiley & Sons, 1967.

[13] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[14] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In *Peer-to-Peer Systems II*, pages 80–87. Springer, Berlin / Heidelberg, 2003.

[15] 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.

[16] Christian Cachin and Stefano Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. In *International Conference on Dependable Systems and Networks (DSN)*, pages 115–124, 2006.

[17] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Third Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, Louisiana, February 1999. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS.

[18] 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.

[19] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2009.

[20] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for Byzantine

fault tolerance. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementations (OSDI)*, Seattle, Washington, November 2006.

[21] Flavin Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34:56–78, February 1991.

[22] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[23] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[25] 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.

[26] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.

[27] Alexandros G. Dimakis, Kannan Ramchandran, Yunnan Wu, and Changho Suh. A survey on network codes for distributed storage. *CoRR*, abs/1004.4438, 2010.

[28] Petros Drineas and Yiorgos Makris. Spare: Selective partial replication for concurrent fault detection in fsms. In *Proceedings of the 16th International Conference on VLSI Design*, VLSID '03, pages 167–, Washington, DC, USA, 2003. IEEE Computer Society.

[29] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34:375–408, September 2002.

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

[31] R. Gallager. Low-density parity-check codes. *IEEE Transactions on Information Theory*, 8(1):21–28, January 1962.

[32] Shuhong Gao. A new algorithm for decoding Reed-Solomon codes. In *in Communications, Information and Network Security, V.Bhargava, H.V.Poor, V.Tarokh, and S.Yoon*, pages 55–68. Kluwer, 2002.

[33] Vijay K. Garg. Implementing fault-tolerant services using state machines: beyond replication. In *Proceedings of the 24th international conference on Distributed computing*, DISC '10, pages 450–464, Berlin, Heidelberg, 2010. Springer-Verlag.

[34] Vijay K. Garg and Vinit Ogale. Fusible data structures for fault-tolerance. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, ICDCS '07, pages 20–, Washington, DC, USA, 2007. IEEE Computer Society.

[35] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[36] Venkatesan Guruswami. *List Decoding of Error-Correcting Codes: Winning Thesis of the 2002 ACM Doctoral Dissertation Competition (Lecture Notes in Computer Science).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[37] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *IN PROC. OF NSDI*, 2005.

[38] Richard Hamming. Error-detecting and error-correcting codes. In *Bell System Technical Journal*, volume 29(2), pages 147–160, 1950.

[39] J. Hartmanis and R. E. Stearns. *Algebraic structure theory of sequential machines (Prentice-Hall international series in applied mathematics).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1966.

[40] Ahmed Helmy, Deborah Estrin, and Sandeep K. S. Gupta. Systematic testing of multicast routing protocols: Analysis of forward and backward search techniques. *CoRR*, cs.NI/0007005, 2000.

[41] James Hendricks, Gregory R. Ganger, and Michael K. Reiter. Verifying distributed erasure-coded data. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 163–168. ACM Press, 2007.

[42] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ASPLOS-IX, pages 93–104, New York, NY, USA, 2000. ACM.

[43] Philip Hingston. Using finite state automata for sequence mining. In *Proceedings of the twenty-fifth Australasian conference on Computer science - Volume 4*, ACSC '02, pages 105–110, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[44] John E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.

[45] David A. Huffman. The synthesis of sequential switching circuits. Technical report, Massachusetts, USA, 1954.

[46] Oliver Kasten and Kay Römer. Beyond event handlers: programming wireless sensors with attributed state machines. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, pages 7+, Piscataway, NJ, USA, 2005. IEEE Press.

[47] Valerie King and Jared Saia. Breaking the $O(n^2)$ bit barrier: scalable Byzantine agreement with an adaptive adversary. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 420–429, New York, NY, USA, 2010. ACM.

[48] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: an ar-

chitecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, November 2000.

[49] Avinash Lakshman and Prashant Malik. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 5–5, New York, NY, USA, 2009. ACM.

[50] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

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

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

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

[54] Minh-Hoang Le, Tu-Bao Ho, and Yoshiteru Nakamori. Detecting citation types using finite-state machines. In *Proceedings of the 10th Pacific-Asia conference on Advances in Knowledge Discovery and Data Mining*, PAKDD'06, pages 265–274, Berlin, Heidelberg, 2006. Springer-Verlag.

[55] David Lee and Mihalis Yannakakis. Closed partition lattice and machine decomposition. *IEEE Trans. Comput.*, 51(2):216–228, 2002.

[56] Derek Leong, Alexandros G. Dimakis, and Tracey Ho. Distributed storage allocation for high reliability. In *ICC*, pages 1–6. IEEE, 2010.

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

[58] Michael Luby. LT codes. In *Proceedings of the 43rd Symposium on Foundations of Computer Science*, FOCS '02, page 271, Washington, DC, USA, 2002. IEEE Computer Society.

[59] Michael Luby, Michael Mitzenmacher, Mohammad Amin Shokrollahi, and Daniel A. Spielman. Efficient erasure correcting codes. *IEEE Transactions on Information Theory*, 47(2):569–584, 2001.

[60] 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.

[61] David J.C. MacKay and Radford M. Neal. Near shannon limit performance of low density parity check codes. *Electronics Letters*, 32:1645–1646, 1996.

[62] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Trans. Comput.*, 37(2):160–174, February 1988.

[63] 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.

[64] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Dag-aware aig rewriting: A fresh look at combinational logic synthesis. In *In DAC*

06: Proceedings of the 43rd annual conference on Design automation, pages 532–536. ACM Press, 2006.

[65] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[66] 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.

[67] Frédérique E. Oggier and Anwitaman Datta. Byzantine fault tolerance of regenerating codes. *CoRR*, abs/1106.2275, 2011.

[68] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in dram. In *SIGOPS OSR*. Stanford InfoLab, 2009.

[69] Dimitris S. Papailiopoulos, Jianqiang Luo, Alexandros G. Dimakis, Cheng Huang, and Jin Li. Simple regenerating codes: Network coding for cloud storage. *CoRR*, abs/1109.0264, 2011.

[70] Rubin A. Parekhji, G. Venkatesh, and Sunil D. Sherlekar. A methodology for designing optimal self-checking sequential circuits. In *Proceedings of the IEEE International Test Conference on Test: Faster, Better, Sooner*, pages 283–291, Washington, DC, USA, 1991. IEEE Computer Society.

[71] Rubin A. Parekhji, G. Venkatesh, and Sunil D. Sherlekar. Concurrent error detection using monitoring machines. *IEEE Des. Test*, 12(3):24–32, September 1995.

[72] 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.

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

[74] 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.

[75] 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.

[76] James S. Plank and Lihao Xu. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In *Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications*, pages 173–180, Washington, DC, USA, 2006. IEEE Computer Society.

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

[78] Parameswaran Ramanathan, Kang G. Shin, and Ricky W. Butler. Fault-tolerant clock synchronization in distributed systems. *Computer*, 23(10):33–42, October 1990.

[79] 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.

[80] T. J. Richardson and R. L. Urbanke. Efficient encoding of low-density parity-check codes. *Information Theory, IEEE Transactions on*, 47(2):638–656, August 2002.

[81] Ron Roth. *Introduction to Coding Theory*. Cambridge University Press, March 2006.

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

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

[84] Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 827–838, New York, NY, USA, 2004. ACM.

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

[86] Swaminathan Sivasubramanian, Michal Szymaniak, Guillaume Pierre, and Maarten van Steen. Replication for web hosting systems. *ACM Comput. Surv.*, 36(3):291–334, 2004.

[87] Ion Stoica, Robert Morris, David Karger, Frans M. Kaashoek, and Hari. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications, 2001.

[88] Madhu Sudan. List decoding: Algorithms and applications. *SIGACT News*, 31:2000, 2000.

[89] 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.

[90] R. M. Tanner, D. Sridhara, A. Sridharan, T. E. Fuja, and D. J. Costello. LDPC block and convolutional codes based on circulant matrices. *Information Theory, IEEE Transactions on*, 50(12):2966–2984, 2004.

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

[92] Hakim Weatherspoon, Tal Moscovitz, and John Kubiatowicz. Introspective Failure Analysis: Avoiding Correlated Failures in Peer-to-Peer Systems. *Reliable Distributed Systems, IEEE Symposium on*, 0:362–367, 2002.

[93] Yunnan Wu and Alexandros G. Dimakis. Reducing repair traffic for erasure coding-based storage via interference alignment. In *Proceedings of the 2009 IEEE international conference on Symposium on Information Theory - Volume 4*, ISIT'09, pages 2276–2280, Piscataway, NJ, USA, 2009. IEEE Press.

[94] Ming-Ming Xiao and Shun-Zheng Yu. Learning automata representation of network protocol by grammar induction. In *Proceedings of the 2010 international conference on Web information systems and mining*, WISM'10, pages 220–227, Berlin, Heidelberg, 2010. Springer-Verlag.

[95] Saeyang Yang. Logic synthesis and optimization benchmarks user guide version 3.0, 1991.

[96] Hiroshi Youra, Tomoo Inoue, Toshimitsu Masuzawa, and Hideo Fujiwara. On the synthesis of synchronizable finite state machines with partial scan. *Systems and Computers in Japan*, 29(1):53–62, 1998.

[97] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

Index

Vita

Bharath Balasubramanian was born in Chennai, India, to B. Sudha and R. Balasubramanian. He received his Bachelors degree in Engineering, Electronics, from Mumbai University in 2004. He received his Master of Science degree in Computer Engineering, from the University of Texas at Austin in 2007. His areas of interest include: Concurrent and Distributed Algorithms, Fault tolerant Distributed Systems, Distributed Storage and Distributed Debugging.

Permanent address: B 1002, Jasmine Towers,
Vasanth Vihar, Thane (West)-400 610,
Maharashtra, India.

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.