Fusion-based DFSMs for Fault Tolerance in Distributed Systems

Bharath Balasubramanian and Vijay K. Garg*
Parallel and Distributed Systems Laboratory,
Dept. of Electrical and Computer Engineering,
The University of Texas at Austin,
Austin, TX 78712, Ph: 512 2398104, Fax: 512 4715120,
Email: bbharath@mail.utexas.edu, garg@ece.utexas.edu

Abstract

Replication is a standard solution for fault-tolerance in distributed systems modeled as deterministic finite state machines (DFSMs or machines). To correct f crash faults among n machines, replication requires nf additional backup machines. We challenge this approach and present a fusion-based solution that requires just f additional backup machines (called fusions or fused backups). In this paper, we first propose a fundamental problem regarding DFSMs, independent of fault tolerance, that has not been explored in the literature so far: Given a machine M, with a set of states and a set of events, can we *replace* it with machines each containing fewer events than M? To formalize this we define a (k,e)-event decomposition of a given machine M, that is a set of k machines each with at least e events fewer than the event set of M, that acting in parallel, are equivalent to M. We present an algorithm to generate such machines with time complexity polynomial in the number of states in M. Second, we use our event decomposition algorithm to generate fused backups that can correct faults among a given set of machines. We show that these backups are *minimal* w.r.t the number of states they contain and the number of events in their event set. Third, we use the notion of *locality sensitive hashing* to present algorithms for the detection and correction of faults for the fusion-based solution. The algorithm for the detection of Byzantine faults has time complexity $O(nf)$ on average, which is the same as that for replication. The algorithm for the correction of both crash and Byzantine faults has time complexity $O(n\rho f)$ with high probability (w.h.p), where ρ is the average state reduction achieved by fusion. We show that for small values of n (for most practical systems, $n < 10$) and ρ (average value of $\rho < 2$ in our experiments), this results in almost no overhead as compared to replication. Finally, we evaluate fusion on the widely used MCNC'91 benchmarks for DFSMs and 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%).

Keywords: Distributed Systems, Fault Tolerance, Finite State Machines.

I. Introduction

Distributed applications often use deterministic finite state machines (or just *machines*) to model computations such as regular expressions for pattern detection, syntactical analysis of documents or mining algorithms for large data sets. These machines executing on distinct distributed processes are often prone to faults. Traditional solutions to this problem involve some form of replication, in which to correct f crash faults [20] among n given machines (referred to as *primaries*), f copies of each primary are maintained [13], [22], [21]. If the backups start from the same initial state as the corresponding primaries and act on the same events, then in the case of faults, the state of the failed machines can be recovered from one of the remaining copies. These backups can also correct $\lfloor f/2 \rfloor$ Byzantine faults [14], where the processes lie about the state of the machine, since a majority of truthful machines is always available. This approach is expensive both in terms of the total number of backup machines, nf and the total backup state space.

*supported by the NSF CNS-0718990, Texas Education Board Grant 781, and Cullen Trust for Higher Education Endowed Professorship.

Consider a distributed application that is searching for three different string patterns among a huge file, as modeled by the state machines A, B and C shown in Fig. 1. A state machine in our system consists of a finite set of states (including the initial execution state) and a finite set of events. On application of an event, the state machine transitions to the next state based on the state-transition function. For example, machine A in Fig. 1 contains the states $\{a^0, a^1\}$, events $\{0, 2\}$ and the initial state, shown by the dark ended arrow, is a^0. The state transitions are shown by the arrows from one state to another. Hence, if A is in state a^0 and event 0 is applied to it, then it transitions to state a^1. In this example, A checks the parity of $\{0, 2\}$ and so, if it is in state a^0, then an even number of 0s or 2s have been applied to the machine and if it is in state a^1, then an odd number of the inputs have been applied. Machines B and C check for the parity of $\{1, 2\}$ and $\{0\}$ respectively.

To correct one crash fault among these machines, replication requires a copy of each of them, resulting in three backup machines, consuming total state space of eight (2^3). Rather than replicate the machines, we can correct one fault by maintaining just one additional machine F_1 shown in Fig. 1. The relevant events from the client (or environment) are applied to all the machines. So if the event sequence 0, 0, 1, 2 is applied on all the machines, A, B, C and F_1 will be in states a^1, b^0, c^0 and f_1^1 respectively. Assume a crash fault in C. Given the parity of 1s (state of F_1) and the parity of 1s or 2s (state of B), we can first determine the parity of 2s. Using this, and the parity of 0s or 2s (state of A), we can determine the parity of 0s (state of C). Hence, we can correct the crash fault in C using A, B and F_1. This argument can be extended to correcting one fault among any of the machines in $\{A, B, C, F_1\}$. This approach consumes fewer backups than replication (one vs. three) and less backup state space (two vs. eight).

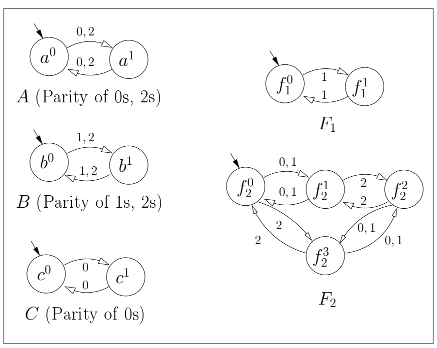


Fig. 1. Fused-Backups for Fault Tolerance

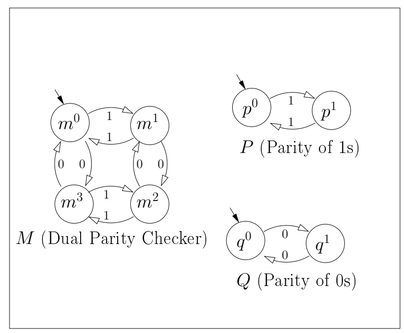


Fig. 2. Event-based Decomposition

However, it is not always possible to design these backups merely by inspection. In Fig. 1, it may not be obvious that F_1 and F_2 can correct two crash faults among the primaries. In [17], we present the theory and algorithm to automatically generate f backup machines (called *fusions*) for any given set of primaries that can correct f crash faults (or $\lfloor f/2 \rfloor$ Byzantine faults). In this paper, we focus on the three main challenges faced by fusion which are the large event-sets of the fusions, the high time complexity for the generation of fusions and the high cost for detecting and correcting faults. To summarize our contributions in this paper:

a) Event-based Decomposition: We start with 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. 2, 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 [10], [15] and the minimization of completely specified machines [12], [11], this is the first paper that presents the problem of event-reduction.

In this paper, 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 a decomposition with time complexity polynomial in the number of states 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.

b) Space-Event Optimized Fusion Algorithm: We apply our event-decomposition algorithm to generate backups for fault tolerance that are optimized for both events and states. In Fig. 1, it is better to choose the 1-event F_1 over the 3-event F_2 as a backup machine to correct one fault. We show that if our solution achieves no event-reduction, then no solution with the same number of backups achieves it. Further, we present an incremental approach for generating the fusions that improves the time complexity by a factor of ρ^n, where ρ is the average state savings achieved by fusion.

c) Efficient Algorithms for Detection/Correction of Faults: In [17], the algorithm for the correction of crash and Byzantine faults, has time complexity $O(n^2\rho + n\rho f + s^n)$, where n is the number of primaries, f is the number of crash faults, s is the maximum number of states among primaries and ρ is the average state savings achieved by fusion. In this paper, 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. The time complexity for crash and Byzantine correction in replication is $\theta(f)$ and $O(nf)$ respectively. Hence, for small values of n and ρ, fusion causes almost no overhead for recovery. In Table I we summarize the notation used in this paper and in Table II we compare replication and the current version of fusion.

TABLE I
SYMBOLS/NOTATION USED IN THE PAPER

$\mathcal{P}$	Set of primaries	n	Number of primaries
RCP	Reachable Cross Product of $\mathcal{P}$	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	Event-Reduction parameter

TABLE II
FUSION VS. REPLICATION (n PRIMARIES, $O(s)$ STATES EACH, f FAULTS, $|\Sigma|$ TOTAL EVENTS, AVERAGE STATE REDUCTION ρ)

	Replication	**Fusion**
Number of Backups	nf	f
Backup Space	$O(s^{nf})$	$O((s/\rho)^{nf})$
Backup Generation Time Complexity	$O(nsf)$	$O(s^n\|\Sigma\|f/\rho^n)$
Maximum Events/Backup	Maximum Events/primary	Minimal for f backups
Byzantine Detection Time Complexity	$O(nf)$	$O(nf)$ on average
Crash Correction Time Complexity	$\theta(f)$	$O(n\rho f)$ w.h.p
Byzantine Correction Time Complexity	$O(nf)$	$O(n\rho f)$ w.h.p

d) Evaluation of Fusion: In [17], we evaluated fusion on simple examples such as counters and dividers. In this paper, we evaluate our fusion algorithm on the MCNC'91 [23] 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%. To illustrate the practical use of fusion, we apply its design to the *grep* application of the MapReduce framework [5]. Using a simple example, we show that the currently used checkpointing approach for fault tolerance needs 600,000 map tasks causing high latency, while replication demands 1200,000 tasks with minimum latency. Fusion offers the best compromise with just 800,000 tasks but smaller latency than the checkpointing approach.

II. Model

The DFSMs in our system execute on separate processes with no shared state or communication. Clients of the state machines issue the events (or commands) to the concerned primaries and backups, all of which act on them in the same relative order. We assume loss-less FIFO communication links with a strict upper bound on the time taken for message delivery. Faults in our system are of two types: crash faults, resulting in a loss of the execution state of the machines and Byzantine faults resulting in an arbitrary execution state. Henceforth in the paper, when we simply say faults, we refer to crash faults. When faults are detected by a trusted recovery agent using timeouts (crash faults) or a detection algorithm (Byzantine faults) no further events are sent by any client to these machines. After the machines act on all events sent to them thus far, the recovery agent obtains their states, and recovers the correct execution states of all faulty machines. Since we assume a trusted recovery agent, the work on consensus in the presence of Byzantine faults [6], [19], does not apply to our paper. In the following section, we summarize the relevant concepts and results introduced in our previous work.

III. Background [17]

1) State-based Decomposition: A DFSM, denoted by R, consists of a set of states X_R, set of events Σ_R, transition function $\alpha_R : X_R \times \Sigma_R \rightarrow X_R$ and initial state x_R^0. The size of R, denoted by $|R|$ is the number of states in R. We can partition the state space of R such that the transition function α_R, maps each block of the partition to another block for all events in Σ_R [10], [15]. In other words, we combine the states of R to generate machines that are consistent to the transition function. The set of all machines generated by combining the states of R is called the *closed partition set* of R (example in Fig. 3).

Consider machine M_2 in Fig. 3, generated by combining the states r^0 and r^2 of R. On event 0, $\{r^0, r^2\}$ self-transitions to $\{r^0, r^2\}$ (self transitions not shown). However, since r^0 and r^2 transition to r^1 and r^3 respectively on event 1, we need to combine the states r^1 and r^3. Continuing this procedure, we obtain the combined states in M_2. 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). P and Q are incomparable, i.e., $P \| Q$, if $P \nleq Q$ and $Q \nleq P$. In Fig. 3, $F_1 < M_2$, while $M_1 \| M_2$.

2) Minimum Hamming distance for DFSMs (d_{min}): Consider a set of machines $\mathcal{R}$ each less than R, i.e., machines belonging to the closed partition set of R. We define the Hamming distance [9] between each $r^i, r^j \in X_R$, denoted $d(r^i, r^j)$, as the number of machines in $\mathcal{R}$ that contain r^i and r^j in different blocks (*separate* r^i and r^j). The minimum Hamming distance across all such pairs is denoted $d_{min}(\mathcal{R})$ or just d_{min}. In Fig. 3, if $\mathcal{R} = \{A, B\}$, $d(r^0, r^1) = 1$ (B separates them), while $d(r^0, r^7) = 0$ and hence $d_{min} = 0$.

Given the state of the machines in $\mathcal{R}$ we can determine the state of R if there is at least one machine in $\mathcal{R}$ to distinguish between each pair of states in X_R, or in other words, $d_{min} > 0$. In Fig. 3 if $\mathcal{R} = \{A, B\}$ and A and B are in states $a^0 = \{r^0, r^1, r^7, r^6\}$ and $b^0 = \{r^0, r^2, r^7, r^5\}$, we cannot determine if R is in state r^0 or r^7 (intersection of a^0 and b^0). However, if $\mathcal{R} = \{A, B, C\}$ ($d_{min} = 1$), then given that A, B and C are in a^0, b^0 and c^0, we can determine that R is in state r^0 (only state common to a^0, b^0 and c^0).

3) Fault Tolerance in DFSMs: To generate the backups (or fusions) for a set of machines, we first construct their *reachable cross product*. Given any two machines $A = (X_A, \Sigma_A, \alpha_A, x_A^0)$ and $B = (X_B, \Sigma_B, \alpha_B, x_B^0)$, their reachable cross product, denoted RCP($\{A, B\}$) is the machine which consists of all the states in the product set of X_A and X_B reachable from the initial state $\{x_A^0, x_B^0\}$, with the transition function $\alpha_{RCP}(\{a, b\}, \sigma) = \{\alpha_A(a, \sigma), \alpha_B(b, \sigma)\}$ for all reachable states $\{a, b\} \in X_A \times X_B$ and $\sigma \in \Sigma_A \cup \Sigma_B$. Given a set of n primaries $\mathcal{P}$, their reachable cross product is denoted RCP $(X_{RCP}, \Sigma, \alpha_{RCP}, r^0)$, where Σ is the union of the event sets of all primary machines. The machine R in Fig. 3, is in fact the RCP of $\mathcal{P} = \{A, B, C\}$ shown in Fig. 1. For convenience, we label the states of the RCP, $r^0 \ldots r^7$, where each $r^i \in X_{RCP}$ is a tuple consisting of the primary states (mapping shown in Fig. 3). The closed partition set of the RCP always includes the primary machines and its states correspond to the RCP states that contains it. In Fig. 3, $a^0 = \{a^0b^0c^0, a^0b^1c^0, a^0b^1c^1, a^0b^0c^1\}$.

Given the state of the RCP, the state of the primaries can be determined. The basic goal of fault tolerance is to generate a set of machines $\mathcal{F}$, each less than the RCP, so that despite f crash faults, there are sufficient

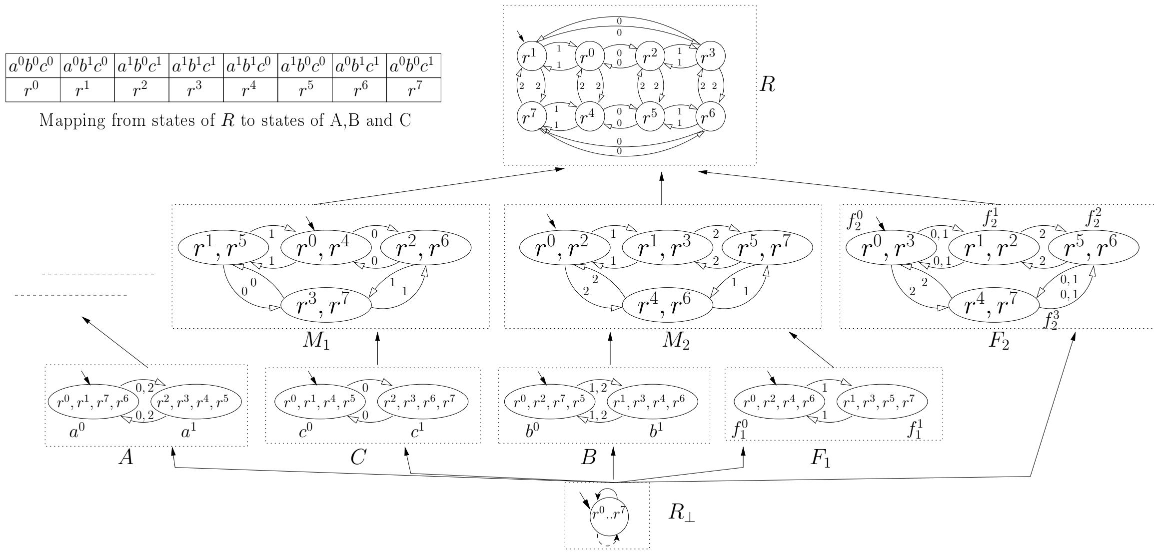


Fig. 3. Set of Machines less than R (all machines not shown due to space constraint)

machines in $\mathcal{P} \cup \mathcal{F}$, i.e., among the primaries and backups, whose $d_{min} > 0$. In other words, a set of machines in $\mathcal{P} \cup \mathcal{F}$ can correct f crash faults iff $d_{min}(\mathcal{P} \cup \mathcal{F}) > f$. In Fig. 3, for $\mathcal{P} = \{A, B, C\}$ and $\mathcal{F} = \{F_1, F_2\}$, it can be seen that $d_{min}(\mathcal{P} \cup \mathcal{F}) > 2$. Consider the state of the machines after the application of the event sequence 0, 1, 1 on the machines in $\mathcal{P} \cup \mathcal{F}$. Assume that B and C crash and we need to recover their state. Given the state of A, F_1 and F_2 as $a^1 = \{r^2, r^3, r^4, r^5\}$, $f_1^0 = \{r^0, r^2, r^4, r^6\}$ and $f_2^1 = \{r^1, r^2\}$, we can determine the state of the RCP as r^2 (only state common to a^1, f_1^0 and f_2^1). Since $r^2 = a^1 b^0 c^1$, we can recover the states of B and C as b^0 and c^1 respectively.

When $|\mathcal{F}| = f$, we call it the f-fusion of $\mathcal{P}$ and call the machines in $\mathcal{F}$, fused-backups or just *fusions*. An f-fusion is *minimal* if there exists no other f-fusion $\mathcal{G}$ in which every machine is less than or equal to some machine in $\mathcal{F}$ and at least one machine is strictly less than some machine in $\mathcal{F}$. In section VI, we describe how an f-fusion can also detect f Byzantine faults or correct $\lfloor f/2 \rfloor$ Byzantine faults.

Coding theory is often used in data fault tolerance for reducing redundancy [18], [4]. In our previous work, we present coding-theoretic solutions to fault tolerance in data structures [2] and infinite state machines [7]. However, a direct coding-theoretic approach to DFSMs, in which we maintain the parity of the states of each machine would be too expensive in terms of communication and computation, since after every event transition, the machine needs to sends its state and the parity needs to be recalculated. Instead, we use our Hamming distance metric to construct backups that independently act on events.

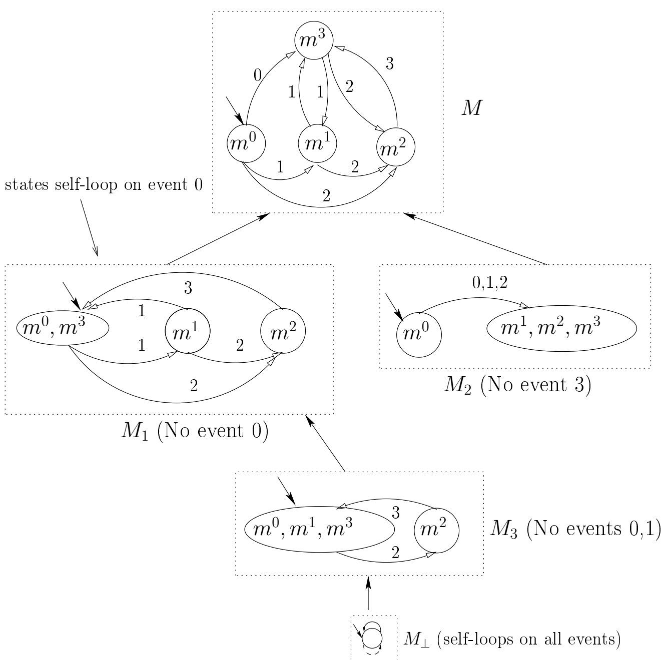
IV. Event-Based Decomposition of Machines

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 1: 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 $d_{min}(\mathcal{E}) > 0$ and $\forall P(X_P, \alpha_P, \Sigma_P, p^0) \in \mathcal{E}$, $|\Sigma_P| \leq |\Sigma_M| - e$.

As $d_{min}(\mathcal{E}) > 0$, given the state of the machines in $\mathcal{E}$, the state of M can be determined (section III-2). So, the machines in $\mathcal{E}$, each containing at most $|\Sigma_M| - e$ events, can effectively replace M. In Fig. 4, 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 (according to the order $\leq$ defined in III-1) 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 to generate the set of largest machines less than P containing at least one event less than Σ_P. To generate a machine less than 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. 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 ensuring that $d_{min}(\mathcal{E}) > 0$ [1].



`eventDecompose`
Input: Machine M $(X_M, \alpha_M, \Sigma_M, m^0)$, e;
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
$E \leftarrow$ Any machine in $\mathcal{M}$ separating (m_i, m_j);
if $(E == \{\})$ **return** $\{\}$;
else $\mathcal{E} \leftarrow \mathcal{E} \cup \{E\}$;
return $\mathcal{E}$;

`reduceEvent`
Input: Machine P $(X_P, \alpha_P, \Sigma_P, p^0)$;
Output: Largest Machines $< P$ with $\leq |\Sigma_P| - 1$ events;
$\mathcal{B} = \{\}$;
for $(\sigma \in \Sigma_P)$
Set of states, $X_B = X_P$;
for $(s \in X_B)$ //combine states to self-loop on σ
$s = s \cup \alpha_P(s, \sigma)$;
$\mathcal{B} = \mathcal{B} \cup \{$Largest machine consistent with $X_B\}$;
return Incomparable machines in $\mathcal{B}$;

Fig. 4. Event-based Decomposition

Let the 4-event machine M shown in Fig. 4 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

[1]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 till no pair remains.

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. We show in appendix A that the `eventDecompose` algorithm has time complexity $O(|X_M|^3|\Sigma_M|^e)$ and also present the proof for the following theorem.

Theorem 1: 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 $k \le |X_M|^2$.

V. State-Event Optimized Fusions

```
genFusion
  Input: Primaries 𝒫, faults f, event depth β;
  Output: f-fusion of 𝒫;
  ℱ ← {};
  for (i = 1 to f)  //Loop 1
    ℳ ← {RCP(𝒫)};
    for (j = 1 to β) //Loop 2
      𝒢 ← {};
      for (M ∈ ℳ)
        𝒢 = 𝒢 ∪ reduceEvent(M);
      ℳ = Machines in 𝒢 that increment d_min;
    M ← Any machine in ℳ;
    while (M ≠ RCP(𝒫)_⊥) //Loop 3
      C ← reduceState(M);
      M= Machine in C that increments d_min;
    ℱ ← {M} ⋃ ℱ;
    return ℱ;
```

```
reduceState
  Input: Machine P (X_P, α_P, Σ_P, p^0);
  Output: Largest Machines with ≤ |X_P| − 1 states;
  ℬ = {};
  for (s_i, s_j ∈ X_P)
    //combine states s_i and s_j
    Set of states, X_B = X_P with (s_i, s_j) combined;
    ℬ = ℬ ∪ {Largest machine consistent with X_B};
  return Incomparable machines in ℬ;
```

```
incFusion
  Input: Primaries 𝒫, faults f, event depth β;
  Output: f-fusion of 𝒫;
  ℱ ← {};
  for each (P_i ∈ 𝒫)
    ℱ ← genFusion({P_i} ∪ RCP(ℱ), f, β);
  return ℱ;
```

Fig. 5. Optimized Fusion Algorithm

Given a set of n primaries $\mathcal{P}$, we present an algorithm in [17] to generate a minimal f-fusion of $\mathcal{P}$. In this paper, we present an algorithm to generate fusions that are optimized for both states and events. We show that if each fusion in our solution contains more than $\Sigma - \beta$ events, then no f-fusion of $\mathcal{P}$ contains a machine with less than or equal to $\Sigma - \beta$ events, where β is a user defined parameter. Further, we present an incremental approach to this problem that improves the time complexity by a factor of ρ^n, where ρ is the average state reduction achieved by fusion, i.e., (|RCP|/Average size of a fusion).

The `genFusion` algorithm that generates the fusion machines is shown in Fig. 5. Starting with the RCP of the primaries, $RCP(\mathcal{P})$, the algorithm generates one machine for each iteration of Loop 1 that increases d_{min} by 1 and at the end of f iterations we have f machines in $\mathcal{F}$ such that $d_{min}(\mathcal{P} \cup \mathcal{F}) > f$. Loops 2 and 3 reduce the events and states of the fusion machines.

Loop 2, Event Reduction: Starting with the RCP, which always increases d_{min} by one, Loop 2 uses the `reduceEvent` algorithm in Fig. 4 to iteratively generate reduced event machines that increase d_{min} by one. In each iteration of Loop 2, we generate the set of machines that contain one event less than the machines in the previous iteration and increase d_{min} by one. At the end of β iterations, we generate machine M that increases d_{min} by one and contains at most $\Sigma - \beta$ events, if such a machine exists. At any stage, if no valid machine was found, we exit the loop and select a machine from the previous iteration.

Loop 3, State Reduction [17]: In Loop 3, we try to find a minimal machine less than the event-reduced M that increases d_{min} by one. Starting with M, the `reduceState` algorithm in Fig. 5 generates the set of largest machines less than M in which at least two states of M are combined. We choose a machine in that set that increases d_{min} and reduce it until no further state reduction is possible (hit the bottom machine $RCP(\mathcal{P})_\perp$).

In Fig. 3, let $\mathcal{P} = \{A, B, C\}, f = 1, \beta = 2$. Since, $d_{min}(\mathcal{P}) = 1$, we need to add a machine that increases d_{min} to two. The set of machines containing one event less than the RCP are M_1 and M_2 among which only M_2 increases d_{min}. Reducing the event-set of M_2, at the end of $\beta = 2$ iterations, $M = F_1$. Since there is no machine less than F_1 that increases d_{min}, no state reduction is possible and the `genFusion` algorithm returns F_1. Note that, for $\beta = 0$ (no event-reduction), the `genFusion` algorithm is identical to the one in [17]. However, without event-reduction, the state reduction algorithm can combine r^0 and r^3 into a single block and generate F_2 as the largest machine containing this block. Since this is a minimal machine, the `genFusion` algorithm can return this 3-event machine. The event-reduction in the current version forces the algorithm to pick the 1-event machine F_1. In appendix B, we show that the time complexity of `genFusion` is $O(N^2|\Sigma|^{\beta} f + N^3|\Sigma| f)$, where $N = |\text{RCP}|$ and present a proof for the following theorem.

Theorem 2: Given a set of n machines $\mathcal{P}$, the `genFusion` algorithm generates a minimal f-fusion (state minimality) of $\mathcal{P}$ such that if each machine in $\mathcal{F}$ contains more than $|\Sigma| - \beta$ events, then no f-fusion of $\mathcal{P}$ contains a machine with less than or equal to $|\Sigma| - \beta$ events (event minimality).

Given n primaries each of size s, the `genFusion` algorithm generates their RCP, that has size $O(s^n)$, and hence the algorithm can have very high execution times. In Fig. 5, 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. In appendix C, we illustrate this approach with an example, present the proof of correctness and show that it has time complexity ρ^n times better than that of the `genFusion` algorithm, where ρ is the average state reduction achieved by fusion.

VI. Detection and Correction of Faults

In [17], the time complexity to detect and correct faults is $O(n^2\rho + n\rho f + N)$, where n is the number of primaries, f is the number of crash faults, s is the size of each machine, N is the size of the RCP and ρ is the average state reduction achieved by fusion. 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. Throughout this section, we refer to Fig. 3, 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 (*primary tuple*) while the state of each backup is represented as the set of primary tuples it corresponds to (*tuple-set*). In Fig. 3, 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 $\{r^0, r^2, r^4, r^6\}$).

A. Detection of Byzantine Faults

Given the primary tuple and the tuple-sets corresponding to the backup states, the `detectByz` algorithm in Fig. 6 detects up to f Byzantine faults (liars). Assuming that the tuple-set of each backup 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. 3, 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 $f_1^1 = \{a^0b^1c^0, a^1b^1c^1, a^1b^0c^0, a^0b^0c^1\}$ or $f_2^1 = \{a^0b^1c^0, a^1b^0c^1\}$. In the following theorem we show that if there are liars in the system, then the primary tuple will not be present in at least one of the backup tuple-sets.

Theorem 3: Given a set of n machines $\mathcal{P}$ and an f-fusion $\mathcal{F}$ corresponding to it, the `detectByz` algorithm detects up to f Byzantine faults among them.

In appendix D we present the proof for this theorem and also show that the space complexity for the `detectByz` algorithm is $O(Nfn \log s)$ while its time complexity 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 replicas, giving time complexity $O(nf)$.

B. Correction of Faults

Given the primary tuple and the tuple-sets of the backup states, to correct f crash faults (or $\lfloor f/2 \rfloor$ Byzantine faults), we first need to find the tuples among the backup tuple-sets that are within Hamming distance of f

```
detectByz
    Input: set of of fusion states B, primary tuple r;
    Output: true or false
    for (b ∈ B)
        if ¬(hash_table(b) · contains(r))
            return false;
    return true;
```

```
correctCrash
    Input: set of of fusion states B, primary tuple r,
    crash faults among the primaries c (≤ f);
    Output: corrected primary n-tuple;
    D ← {} //list of tuple-sets
    for (b ∈ B)
        //tuples in b within Hamming distance c of r
        S ← lsh_tables(b) · search(r, c);
        D · add(S);
    return Intersection of sets in D; // singleton w.h.p
```

```
correctByz
    Input: set of of fusion states B, primary tuple r;
    Output: corrected primary n-tuple;
    D ← {} //list of tuple-sets
    for (b ∈ B)
    //tuples in b within Hamming distance ⌊f/2⌋ of r
        S ← lsh_tables(b) · search(r, ⌊f/2⌋);
        D · add(S);
    G ← Set of tuples that appear in D;
    V ← Vote array of size |G|;
    for (g ∈ G)
        // get votes from fusions
        V[g] ← Number of times g appears in D;
        // get votes from primaries
        for (i = 1 to n)
            if(r[i] ∈ g)
                V[g] + +;
    return Tuple g : V[g] ≥ n + ⌊f/2⌋;
```

Fig. 6. Detection and Correction of Faults

($\lfloor f/2 \rfloor$ for Byzantine faults) from the primary tuple (explained in sections VI-B1 and VI-B2). In Fig. 3, 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], [8]. Based on this, we maintain L hash tables, $\{g_1 \dots g_L\}$, for each fusion state at the recovery agent. The hash function for g_j, takes as input an n-tuple, selects k coordinates uniformly at random from them and returns the concatenated bit representation of these coordinates. In the example shown in Fig. 7(i), 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.

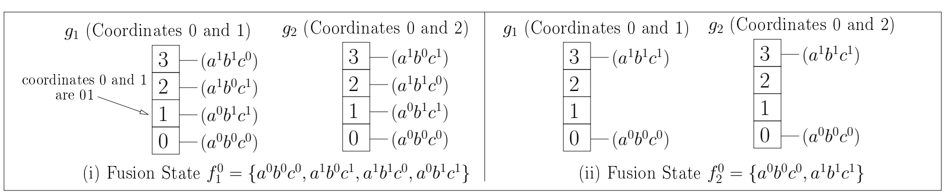


Fig. 7. LSH Example for fusion states in Fig. 3 with $k = 2$, $L = 2$

Given a point q and distance f, we obtain the points found in the buckets $g_j(q)$ for $j = 1 \dots L$, and return those that are within distance of f from q. For example, in Fig. 7(i), given $q = a^0b^1c^0$, $f = 2$, this point hashes into the 1^{st} bucket of g_1 and the 0^{th} bucket of g_2 and hence the points returned are $a^0b^1c^1$ and $a^0b^0c^0$ respectively. If we set $L = \log_{1-\gamma^k} \delta$, where $\gamma = 1 - f/n$, such that $(1 - \gamma^k)^L < \delta$, then any f-neighbor of a point q is returned with probability at least $1 - \delta$ [1], [8]. In the following sections, we present algorithms for the correction of crash and Byzantine faults based on these LSH functions.

1) Crash Correction: Given the primary tuple (with possible gaps because of faults) and the tuple-sets of the available backup states, the `correctCrash` algorithm in Fig. 6 corrects up to f crash faults. The algorithm finds the tuples in the tuple-sets of each fusion state b that are within a Hamming distance c (actual number of faults) of the primary tuple r using the LSH tables for each fusion state. If the intersection of these sets is singleton, then we return that as the correct primary tuple. When the intersection is not singleton, we need to exhaustively search each fusion state for points within distance c of r (LSH has not returned all of them), but this happens with a very low probability [1], [8]. In Fig. 3, 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 $r = a^0\{\}\{\}$ 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 4: Given a set of n machines $\mathcal{P}$ and an f-fusion $\mathcal{F}$ corresponding to it, the `correctCrash` algorithm corrects up to f crash faults among them.

In appendix E, we present the proof for this theorem and show that the space complexity of the `correctCrash` algorithm is $O(Nfn \log s)$ and its time complexity is $O(n\rho f)$ w.h.p. Crash correction in replication simply involves copying the state of the replicas of f failed primaries which has time complexity $O(f)$.

2) Byzantine Correction: Given the primary tuple and the tuple-sets of the backup states, the `correctByz` algorithm in Fig. 6 corrects up to $\lfloor f/2 \rfloor$ Byzantine faults. The 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 votes 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. 3, let the states of machines A, B, C F_1 and F_2 are 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. The algorithm returns $a^0b^0c^0$, with four votes in total (one each from A, C, F_1 and F_2), since $n + \lfloor f/2 \rfloor = 3 + 1 = 4$. We show in the following theorem that there are enough machines separating each pair of tuples and even with liars the true primary tuple will get sufficient votes.

Theorem 5: Given a set of n machines $\mathcal{P}$ and a f-fusion $\mathcal{F}$ corresponding to it, the `correctByz` algorithm corrects up to $\lfloor f/2 \rfloor$ Byzantine faults among them.

In appendix F, we present a proof for the following theorem and show that the space complexity of the `correctByz` algorithm is $O(Nfn \log s)$ and its time complexity of is $O(n\rho f)$ w.h.p. In the case of replication, we just need to obtain the majority across f copies of each primary with time complexity $O(nf)$.

VII. Evaluation

A. Experimental Results

In [17], we evaluate fusion for simple examples such as counters and dividers. In this section, we evaluate fusion using the MCNC'91 benchmarks [23] for DFSMs, widely used for research in the fields of logic synthesis and finite state machine synthesis [16], [24]. We implemented the `incFusion` algorithm of Fig. 5 in Java 1.6 and compared the performance of fusion with replication for 100 different combinations of the benchmark machines, with $n = 3$, $f = 2$, $\beta = 3$ and present some of the results in Table III. The machine descriptions, implementation and detailed results are available in [3].

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 III 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 the total number of primary events, 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 [3] 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.

TABLE III
EVALUATION OF FUSION ON THE MCNC'91 BENCHMARKS

Machines	Replication State Space	Fusion State Space	% Savings State Space	Primary Events	Fusion Events	% Reduction Events
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

B. Practical Example: MapReduce

To motivate the practical use of fusion, we discuss its application to the MapReduce framework which is used to model large scale distributed computations. Typically, the Map-Reduce framework is built using the master-worker configuration where the master assigns the map and reduce tasks to various workers. Due to high cost of resources in replication, handling faults among the map workers is primarily based on checkpointing in which the processes periodically write to permanent storage. In the case of faults, the tasks are restarted from the last available state. This approach increases latency and may be inadequate for some applications.

Consider a distributed grep application over large files, where the master assigns three map tasks, each searching for one of the string patterns modeled by $\{A, B, C\}$ in Fig. 1. When the input files are partitioned into 200,000 chunks of data (the usual number in [5]), the current checkpointing-based approach requires 200,00*3 = 600,000 tasks in total, while causing high latency. A replication-based solution for correcting just one fault will involve creating a replica of each of the tasks A, B and C for each chunk of data, requiring 1200,000 tasks in total. A fusion-based approach needs to run only one additional backup task for each chunk of data, running F_1 shown in Fig. 1. Though recovery is costlier than replication, this approach requires only 800,000 tasks with much better latency than checkpointing.

VIII. Conclusion

We challenge the traditional approach of replication that requires nf backups to correct f crash faults among n machines and present a fusion-based solution that requires only f backups consuming considerably lesser state space. We present a problem that is fundamental to DFSMs: Can we replace a given DFSM with DFSMs containing fewer events? To formalize this, we introduce the concept of a (k,e)-event decomposition of a given machine and present efficient algorithms to generate such a decomposition. Based on this, we describe an algorithm to generate fused backups for a given set of machines that is optimized for both states and events.

Further, we present efficient algorithms to detect and correct faults in a system with fused backups. The algorithm for the detection of Byzantine faults has time complexity $O(nf)$ (on average), which is the same as that for replication. We apply the concept of locality sensitive hashing to the correction of faults and the time complexity for the correction of crash and Byzantine faults is $O(n\rho f)$ w.h.p. For relatively small values of n and ρ, fusion causes almost no overhead for recovery. Finally, we evaluate fusion on standard benchmarks for DFSMs and the results confirm that fusion achieves significant savings in space over replication. The event-reduction algorithm ensures that for many examples, the fused backups contain small event sets. Hence, in addition to our results on the theoretical optimality of the fused backups, we have illustrated the practical usefulness of fusion.

References

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

[2] B. Balasubramanian and V. K. Garg. A fusion-based approach for handling multiple faults in data structures. Technical Report ECE-PDS-2009-001, Parallel and Distributed Systems Laboratory, ECE Dept. University of Texas at Austin, 2009.

[3] B. Balasubramanian and V. K. Garg. Fsm backup library (implemented in java 1.6). In *Parallel and Distributed Systems Laboratory, http://maple.ece.utexas.edu*, 2011.

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

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

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

[7] V. K. Garg. Implementing fault-tolerant services using state machines: Beyond replication. In *International Symposium on Distributed Computing (DISC)*, pages 450–464, 2010.

[8] A. Gionis, P. Indyk, and R. 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.

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

[10] 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.

[11] J. E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.

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

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

[14] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.

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

[16] A. Mishchenko, S. Chatterjee, and R. 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.

[17] V. Ogale, B. Balasubramanian, and V. K. Garg. A fusion-based approach for tolerating faults in finite state machines. *Parallel and Distributed Processing Symposium, International*, 0:1–11, 2009.

[18] D. A. Patterson, G. Gibson, and R. 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.

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

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

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

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

[23] S. Yang. Logic synthesis and optimization benchmarks user guide version 3.0, 1991.

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

Appendix

A. Time complexity, Proof of Correctness for `eventDecompose`

The `reduceEvent` algorithm visits each state of machine M to create blocks of states which loop to the same block on event $\sigma \in \Sigma_M$. This has time complexity $O(|X_M|)$. The cost of generating the largest closed partition corresponding to this block is $O(|X_M||\Sigma_M|)$. 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 time complexity to reduce at most one event from any machine is $O(|X_M||\Sigma_M|^2)$. Hence, the time complexity to generate the set of event-reduced machines in $\mathcal{M}$ at the end of e iterations is $O(|X_M| \cdot |\Sigma_M|^2(|\Sigma_M| + |\Sigma_M|^2 + \ldots +)|\Sigma_M|^e)$ which reduces to $O(|X_M||\Sigma_M|^e)$.

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 complexity of Loop 3 is $O(|X_M|^3|\Sigma|^e)$, which is the overall time complexity of the `eventDecompose` algorithm.

Theorem 1: 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 all 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 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, $k \leq |X_M|^2$. ■

B. Time complexity Analysis, Correctness Proof for `genFusion`

We first analyze the time complexity of Loop 2. For each event, the `reduceEvent` algorithm iterates through all the states in the RCP and forms the largest closed partition corresponding to the set of blocks generated. This has time complexity $O(N|\Sigma|)$, where $N = |RCP|$. Since this is done for all events in Σ, the time complexity of the `genFusion` algorithm is $O(N|\Sigma|^2)$ and the number of machines that it generates is $O(|\Sigma|)$. In each iteration of Loop 2, we check to see if there is a machine that increases d_{min} by one which costs $O(N^2)$ time per machine. The number of machines generated in each iteration increases exponentially in β since we may reduce the event set for each of the machines generated in the previous iteration. So, the cost of Loop 2 is $O((N^2 + N|\Sigma|^2)(1 + |\Sigma| + |\Sigma|^2 + \ldots |\Sigma|^\beta))$, which reduces to $O(N^2|\Sigma|^\beta)$. Loop 3 has time complexity $O(N^3|\Sigma|)$ [17]. The complexity of the loops dominate the complexity of the algorithm. Since there are f iterations of these loops, the time complexity of the `genFusion` algorithm is $O(N^2 f|\Sigma|^\beta + N^3 f|\Sigma|)$.

To prove theorem 2, we refer to relevant concepts introduced in [17]. To generate fusions, we add a machine in each iteration which increases the minimum Hamming distance by one. Each machine added, increases the weight of some pair of states r^i, r^j by one. If a machine M has the states r^i and r^j in distinct blocks, we say that M *covers* the edge (r^i, r^j). It can be seen from the `genFusion` algorithm that every machine added, covers a set of edges, called the edge set of that machine. The edge set of F_j is denoted by E_j. The weight of an edge (r^i, r^j) is $d_{min}(r^i, r^j)$ and the weakest edge is the edge corresponding to the least Hamming distance (d_{min}). We first state a lemma presented in [17].

Lemma 1: Given a set of n machines $\mathcal{P}$, and the set $\mathcal{F}$ returned by the `genFusion` algorithm, let $F_i \in \mathcal{F}$ be the machine returned in the i^{th} iteration. Then, $\forall F_i, F_j \in \mathcal{F} : i < j \Rightarrow E_i \subseteq E_j$.

Proof: If $\mathcal{F}' \subseteq \mathcal{F}$ is the current fusion set during the execution of the `genFusion` algorithm, then the edge set for the next iteration consists of the minimal edges of the machines in $\mathcal{P} \cup \mathcal{F}$. Every time a machine is added to $\mathcal{F}'$, the weights of the edges can increase by at most one and the weight of every minimal edge is incremented by exactly one. Hence, after every iteration the edge set for the next iteration cannot decrease in size. This implies $\forall F_i, F_j \in \mathcal{F} : i < j \Rightarrow E_i \subseteq E_j$. ■

This implies the following two observations.

Observation 1: If an edge e occurs in the edge set of any machine in $\mathcal{F}$ and there are k machines in $\mathcal{F}$ that cover e, then in any valid f-fusion there are at least k machines that cover edge e.

Observation 2: The edge set of the fusion F_1 added in the first iteration of the `genFusion` algorithm has to be a subset of the edge set of all machines part of any valid f-fusion.

We prove theorem 2 based on these observations.

Theorem 2: Given a set of n machines $\mathcal{P}$, the `genFusion` algorithm generates a minimal f-fusion of $\mathcal{P}$ such that if each machine in $\mathcal{F}$ contains more than $|\Sigma| - \beta$ events, then no f-fusion of $\mathcal{P}$ contains a machine with less than or equal to $|\Sigma| - \beta$ events.

Proof:

- $|\mathcal{F}| = f$: Given a set of primaries, we can generate the RCP corresponding to them. Hence, $d_{min}(\mathcal{P}) = 1$. For each iteration of the outer loop, we add one machine to $\mathcal{F}$ that increases $d_{min}(\mathcal{P} \cup \mathcal{F})$ exactly by one. At the end of f iterations we add exactly f fusions such that $d_{min}(\mathcal{P} \cup \mathcal{F}) = f + 1$.
- $\mathcal{F}$ is minimal [17]: Let there be an f-fusion $\mathcal{G} = \{G_1, ..G_f\}$, such that $\mathcal{G}$ is less than f-fusion $\mathcal{F} = \{F_2, F_1, ..., F_f\}$. Hence $\forall j : G_j \le F_j$. 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 `genFusion` algorithm would have returned G_i instead of F_i). From observation 1, if e is covered by k DFSMs 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 $G_j \le F_j$). Hence e can be covered by no more than $k - 1$ in $\mathcal{G}$. This implies that $\mathcal{G}$ is not a valid fusion.
- If each machine in $\mathcal{F}$ contains more than $|\Sigma| - \beta$ events, then no f-fusion can contain a machine with less than or equal to $|\Sigma| - \beta$ events: Let there be an f-fusion $\mathcal{G}$ that contains a machine G with $\le |\Sigma| - \beta$ events. From observation 2, G covers all edges in the edge set E_1, that are covered by $F_1 \in \mathcal{F}$ (the first machine chosen by the `genFusion` algorithm). Since the `genFusion` algorithm could find no machine covering the edges in E_1 at a depth of event reduction $|\Sigma| - \beta$, there cannot exist machines with events less than or equal to $|\Sigma| - \beta$ events that cover E_1. Hence, G cannot exist.

■

C. Time Complexity Analysis, Correctness Proof for `incFusion`

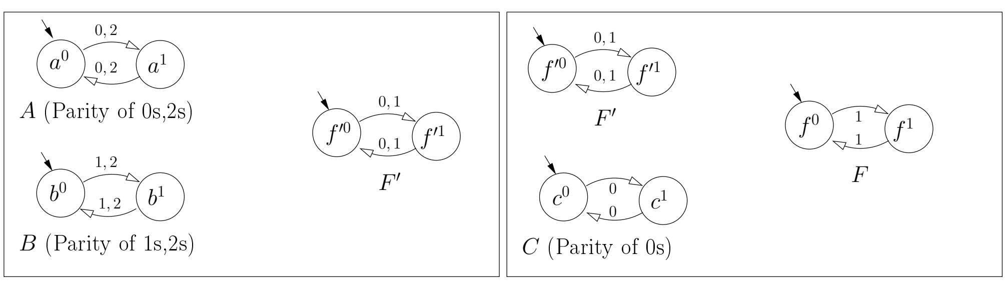


Fig. 8. Incremental Approach: First generate F' and then F

In Fig. 8, 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. Let the number of states in each primary be s. The `genfusion` algorithm has time complexity $O(N^3 \cdot |\Sigma| \cdot f)$ (assuming $\beta = 1$ for simplicity), where N is the size of the cross product of the primaries. For $i = 1$, `genfusion` takes two machines P_1 and P_2 as parameters each of size s. The size of their RCP is $O(s^2)$ and the time complexity for the first iteration is $O(s^6 \cdot |\Sigma| \cdot f)$. Since we assume an average reduction of ρ, the size of each fusion is $O(s/\rho)$. The size of the RCP of these fusions is bound by the size of the RCP of the input machines which is $O(s^2)$ while it has a lower bound of $o(s^2/\rho^2)$. On average, it will be $O(s^2/\rho)$. The input state space for the next iteration is $O(s^3/\rho)$ and hence the time complexity is $O((s^3/\rho)^3 \cdot |\Sigma| \cdot f)$. The input state space for the next iteration is $O(s^4/\rho^2)$. Continuing this analysis, the time complexity of the n^{th} iteration is $O((s^n/\rho^n)^3 \cdot |\Sigma| \cdot f)$. The sum of these terms across all the iterations is a geometric progression dominated by the last term. Hence, the time complexity the `incFusion` algorithm is $O(N^3/\rho^n \cdot |\Sigma| \cdot f)$.

Theorem 3: Given a set of n machines $\mathcal{P}$, the `incFusion` algorithm generates a f-fusion of $\mathcal{P}$.

Proof: We prove it using induction on the iterations of the algorithm.

Base case: For $(i = 1)$, let the f-fusion generated for the primaries $\{P_1, P_2\}$ be denoted $\mathcal{F}^1$. For $(i = 2)$, let the f-fusion generated for $\{P_3, RCP(\mathcal{F}^1)\}$ be denoted $\mathcal{F}^2$. We show that $\mathcal{F}^2$ is a f-fusion of $\{P_1, P_2, P_3\}$. There can only be three cases:

- f machines among $\{P_1, P_2\}$ crash: Since by construction, $\{P_3 \cup RCP(\mathcal{F}^1) \cup \mathcal{F}^2\}$ can correct f crash faults, using the state of $\{P_3 \cup \mathcal{F}^2\}$, we can generate the state of $RCP(\mathcal{F}^1)$. Subsequently, using the state of the remaining 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 $\{P_1, P_2\}$.
- f machines among $\{P_3 \cup \mathcal{F}^2\}$ crash: Since by construction, $\{P_1, P_2 \cup \mathcal{F}^1\}$ can correct f crash faults, using the state of machines in $\{P_1, P_2\}$ we can generate the state of the f machines in $\mathcal{F}^1$. Subsequently, using the state of the remaining machines among $\{P_3 \cup \mathcal{F}^2\}$ and the state of all the machines in $\mathcal{F}^1$ we can generate the crashed machines $\{P_3 \cup \mathcal{F}^2\}$.
- t machines among $\{P_1, P_2\}$ crash $(t > 0)$ and $f - t$ machine among $\{P_3 \cup \mathcal{F}^2\}$ crash $(f - t < f)$: Among the $f + 1$ machines in $\{P_3 \cup \mathcal{F}^2\}$ less than f have crashed. So using the state of the remaining machines, we can generate the state of the machines in $\mathcal{F}^1$ and the state of the crashed machines among $\{P_3 \cup \mathcal{F}^2\}$. Subsequently, using the state of the remaining 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: If the fusion set $\mathcal{F}^i$ generated in iteration i is a f-fusion of $\{P_1 \dots P_{i+1}\}$ and if $\mathcal{F}^{i+1}$ is a f-fusion of $\{P_{i+2}, RCP(\mathcal{F}^i)\}$. To prove: $\mathcal{F}^{i+1}$ is a f-fusion of $\{P_1 \dots P_{i+2}\}$. The proof is similar to that for the base case. If f machines crash among $\{P_1 \dots P_{i+1}\}$, then we can generate the state of the machines in $\mathcal{F}^i$ using the state of the machines among $\{P_{i+2} \cup \mathcal{F}^{i+1}\}$ and then generate the state of the f crashed machines among $\{P_1 \dots P_{i+1}\}$ and $\mathcal{F}^i$. If f machines crash among $\{P_{i+2} \cup \mathcal{F}^{i+1}\}$ then similarly you first generate the state of the machines in $\mathcal{F}^i$ from the state of the remaining machines and then recover the state of the failed machines. The same argument works when the failures are spread across the machines in $\{P_1 \dots P_{i+1}\}$ and $\{P_{i+2} \cup \mathcal{F}^{i+1}\}$. Hence the hypothesis is true. ■

D. Byzantine Detection Complexity Analysis

Theorem 4: Given a set of n machines $\mathcal{P}$ and an f-fusion $\mathcal{F}$ corresponding to it, the `detectByz` algorithm detects up to f Byzantine faults among them.

Proof: When machines lie about their state, we assume that they lie within their state set. For example, in Fig. 3, 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 without a detection algorithm. We show that when there are liars in the system, the primary tuple r will not be present in the tuple-sets of at least one of the fusions.

If r is the correct tuple (without liars), then the liars among the fusions will not contain r in their sets because only one fusion state in each fusion machine contains each primary tuple (fusion states are a partition of the RCP state space). In Fig. 3, if $r = a^1b^0c^1$ (no liars) and F_2 is lying about its state as f_2^0 (truthful state is f_2^1), then r is not present in $f_2^0 = \{a^0b^0c^0, a^1b^1c^1\}$ since it is present in $f_2^1 = \{a^0b^1c^0, a^1b^0c^1\}$.

If r is the incorrect tuple (with liars), then for the fault to go undetected, r must be present in the tuple-sets of all fusion states. The truthful backup tuple-sets will also contain the correct primary tuple $r_{correct}$. Note that, like the fusion states, each primary state can be expressed as a tuple-set that contain the RCP tuples it belongs to. So, the truthful machines among the primaries will also contain $\{r, r_{correct}\}$ in the same tuple-set. For example, in Fig. 3, if $r = a^0b^0c^0$ and $r_{correct} = a^0b^0c^1$, then $a^0 = \{a^0b^0c^0, a^0b^1c^0, a^0b^1c^1, a^0b^0c^1\}$ contains $\{r, r_{correct}\}$ in the same tuple-set. Hence, all the truthful machines (both primaries and backups) contain $\{r, r_{correct}\}$ in the same tuple-set. Since the number of truthful machines is greater than n, at most f machines separate $\{r, r_{correct}\}$. This contradicts the fact that $\mathcal{F}$ is a f-fusion of $\mathcal{P}$ with $d_{min}(\mathcal{P} \cup \mathcal{F}) > f$. ■

Each tuple in a tuple-set of a fusion state contains n states each of size $\log s$, where s is the maximum number of states in any primary. For each fusion, we need to store $O(N)$ such points in the hash table. Hence, the space complexity for storage at the recovery agent is $O(Nfn\log s)$. Since each fusion state is maintained as a hash table at the recovery agent, it can search for the primary tuple in $O(n)$ time, on average. Hence, the time complexity for the `detectByz` algorithm is $O(nf)$ on average.

E. Crash Correction Complexity Analysis

Theorem 5: Given a set of n machines $\mathcal{P}$ and an f-fusion $\mathcal{F}$ corresponding to it, the `correctCrash` algorithm corrects up to f crash faults among them.

Proof: The tuples among the backup tuple-sets within a Hamming distance f of the primary tuple r, are essentially the tuples containing the incomplete r. Since all available fusion-states contain the complete primary tuple, denoted $r_{correct}$, we just need to prove that the intersection of the tuples among the fusion-states containing r is singleton. If not, then there exists at least one other tuple r_{wrong} in all the fusion states containing r. Similar to the proof in theorem 4, since both r_{wrong} and $r_{correct}$ contain r, these tuples will be present in the same tuple-sets of the primaries as well. So, the minimum number of machines containing $\{r_{correct}, r_{wrong}\}$ in the same tuple-set are n (primaries) + available backups, which is greater than or equal to n. Hence, the number of machines separating them is at most f, contradicting $d_{min}(\mathcal{P} \cup \mathcal{F}) > f$. ■

We maintain L hash tables per fusion state, each storing ρ (average number of points in each fusion) n-dimensional points, each containing $\log s$ bits. Since the sum of all such points is N, the total space complexity of storage at the recovery agent is $O(Nfn\log s)$.

Since the number of points per fusion state that are within distance f of the primary tuple r is $O(\rho)$ (average size of the list), the cost of hashing r and retrieving $O(\rho)$ n-dimensional points from $O(f)$ fusion states 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. In order to find the intersection in linear time, we can hash the elements of the smallest set and check if the elements of the other sets are part of this set. The elements found across all sets is the intersection of the sets. 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.

F. Byzantine Correction Complexity Analysis

Theorem 6: Given a set of n machines $\mathcal{P}$ and a 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, $r_{correct}$ will uniquely get $\geq n + \lfloor f/2 \rfloor$ votes. Since there are at most $\lfloor f/2 \rfloor$ liars, $r_{correct}$ will be present in the tuple-sets of at least $n + \lfloor f/2 \rfloor$ truthful machines. Hence the number of votes to $r_{correct}$, $V[r_{correct}] \geq n + \lfloor f/2 \rfloor$. An incorrect primary tuple r_{wrong} can get votes from at most $\lfloor f/2 \rfloor$ liars and the truthful machines that contain both $r_{correct}$ and r_{wrong} in their tuple-sets. Since $d_{min} > f$, among $n + f$ machines, less than n of them contain $\{r_{correct}, r_{wrong}\}$ in the same tuple-set. The number of votes to r_{wrong}, $V[r_{wrong}] < n$ (truthful)+$\lfloor f/2 \rfloor$ (liars) $< n + \lfloor f/2 \rfloor < V[r_{correct}]$. ■

The space complexity analysis is similar to crash correction. The time complexity to generate D, same as that for crash faults, 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 ρ 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 over all time complexity $O(n\rho f)$ w.h.p.