

Reliability Analysis for Unreliable FSM Computations

AMIR HOSSEIN NODEHI SABET, JUNQIAO QIU, and ZHIJIA ZHAO, University of California, Riverside

SRIRAM KRISHNAMOORTHY, Pacific Northwest National Laboratory

Finite State Machines (FSMs) are fundamental in both hardware design and software development. However, the reliability of FSM computations remains poorly understood. Existing reliability analyses are mainly designed for generic computations and are unaware of the special error tolerance characteristics in FSM computations. This work introduces RelyFSM – a state-level reliability analysis framework for FSM computations. By modeling the behaviors of unreliable FSM executions and qualitatively reasoning about the transition structures, RelyFSM can precisely capture the inherent error tolerance in FSM computations. Our evaluation with real-world FSM benchmarks confirms both the accuracy and efficiency of RelyFSM.

CCS Concepts: • **Software and its engineering** → **Software verification**; • **Computer systems organization** → **Reliability**; • **Hardware** → **Fault tolerance**;

Additional Key Words and Phrases: Finite state machine, error tolerance, reliability, probabilistic model

ACM Reference format:

Amir Hossein Nodehi Sabet, Junqiao Qiu, Zhijia Zhao, and Sriram Krishnamoorthy. 2020. Reliability Analysis for Unreliable FSM Computations. *ACM Trans. Archit. Code Optim.* 17, 2, Article 12 (May 2020), 23 pages. <https://doi.org/10.1145/3377456>

1 INTRODUCTION

Finite State Machines (FSMs) are fundamental in both hardware design and software development. At the hardware level, FSMs serve as the underlying computation models for circuit design/logic controllers [3, 26] and memory-based automata processors [1, 11, 33, 51, 52]. At the software level, FSMs are the backbone of many automata-based applications and event-driven systems, such as pattern matching [34, 45], data decoding [27, 49], semi-structured data analytics [6, 18], and network intrusion detection [17, 29].

For their fundamental roles in computing, it is important to understand the reliability of FSM computations in unreliable environments. As hardware manufacturers push the production process to the physical limits, the soft error rates in emerging computer architectures are anticipated to increase [2, 37]. This trend is magnified in approximate computing, where unreliable

This research was supported in part by NSF Award 1565928. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number 66905. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830.

Authors' addresses: A. H. N. Sabet, J. Qiu, and Z. Zhao, University of California, Riverside, 900 University Ave, Riverside, CA, 92521; emails: {anode001, jqiu004}@ucr.edu, zhijia@cs.ucr.edu; S. Krishnamoorthy, Pacific Northwest National Laboratory, P.O. Box 999, MSINJ4-30, Richland, WA, 99352; email: sriram@pnnl.gov.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2020/05-ART12

<https://doi.org/10.1145/3377456>

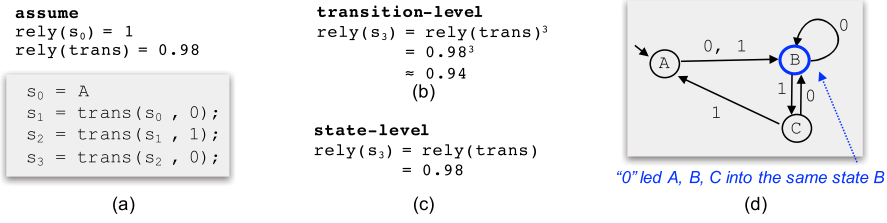


Fig. 1. Transition-level vs. state-level reliability analysis.

hardware components are introduced to simplify the design and improve efficiency [20, 44, 46], such as approximate circuits [12, 53] and approximate storage [42, 47]. As more frequent soft errors are expected on approximate computing platforms, program reliability reasoning frameworks are necessary to ensure that program executions still satisfy the reliability requirement [4, 36, 46].

State of the art. Existing reliability analyses designed for general computations [4, 30, 31, 36] are mainly performed at the instruction level. They assume that errors may occur during the evaluation of individual instructions and infer the reliability of variables by propagating the reliability of individual instructions along program execution paths. This provides a general solution to reason about the program reliability and often yields sufficient accuracies for many applications [4, 46]. However, to apply a similar reliability reasoning for FSM computations, we have to raise the level of abstraction from individual instructions to FSM transitions. Assuming that errors may occur at individual FSM transitions, the reliability of state can be computed by propagating the reliability along the FSM transition trace, referred to as the *transition-level reliability analysis*. As shown in Figure 1(a) and (b), following the transition-level reliability analysis, we can infer that the reliability of s_3 is about 0.94, a joint probability among the reliability of three transitions (0.98^3).

Although the analysis of transition-level reliability is straightforward, its computed value may not faithfully reflect the reliability of the current state that the FSM is in. We demonstrate this with an FSM transition diagram as shown in Figure 1(d) and the reliability reasoning in Figure 1(c). First, notice that in the transition diagram, states A, B, and C all transition to state B after observing the symbol \emptyset . Then, consider the last transition in Figure 1(a). In fact, no matter what state s_2 carries, after consuming the symbol \emptyset , the expected target state would always be state B, assuming the last transition itself is error free. Finally, as the last transition is not error free, the reliability of s_3 turns out to be the same as the reliability of that transition, which is 0.98 rather than 0.94. As the preceding reliability reasoning takes specific states (and their possible transitions) into account, we refer to it as *state-level reliability analysis*.

Essentially, the reliability gap in the preceding example is due to the fact that transition-level reliability analysis is unaware of the potential *error tolerance* in FSM computations. In fact, many FSMs exhibit certain error tolerance capabilities thanks to some common properties in their transition diagrams. In the preceding example, the error tolerance comes from the substructure where all of the states transition to the same state after reading symbol \emptyset —a property known as *state synchronization*. In general, there exist many different ways that errors get tolerated in FSM computations (see Section 2.2). Failing to consider these potential error tolerance cases tends to make reliability characterization less and less precise as the transitions elapse, illustrated in Figure 2.

Overview of this work. To precisely capture the reliability of FSM computations, it is important to reason about the reliability at the *state level* rather than the transition level.¹ This work presents *RelyFSM*, a state-level reliability analysis framework for FSM computations. First, *RelyFSM* allows

¹Despite that the reliabilities of lower-level instructions may still need to be collected.

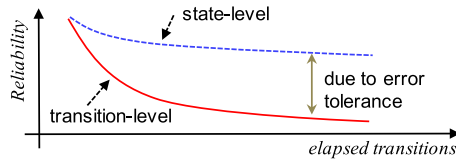


Fig. 2. Reliability gap between transition-level and state-level analysis.

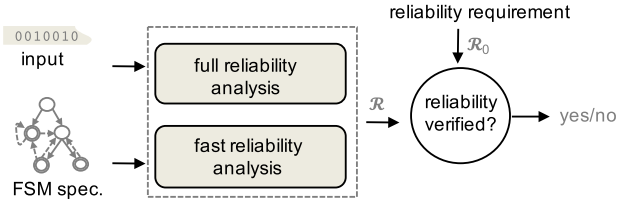


Fig. 3. Workflow of RelyFSM.

users to specify the FSMs and their reliability requirements in a formal way. With the formal specifications, RelyFSM can automatically reason about the reliability of FSM computations and verify it against predefined reliability requirements, as illustrated in Figure 3. Depending on the needs, the reasoning can be conducted either with full respect to the accuracy (i.e., full reliability analysis) or trading off the accuracy for faster analysis (i.e., fast reliability analysis). At the core of these analyses are the mathematical models that can precisely capture the behaviors of unreliable FSM computations with statistical guarantees.

We evaluate RelyFSM with a spectrum of FSM benchmarks drawn from real-world applications. The evaluation shows that the reliability analysis results from RelyFSM are consistent with the ones from extensive fault injection-based approaches, confirming its accuracy. In addition, we also conduct a use case study for RelyFSM on approximate searching in biological databases.

Contributions. This work makes a fourfold contribution:

- To the best of our knowledge, for the first time, this work provides a systematic characterization on the error tolerance properties of FSM computations (Section 2) and a formalization of the FSM reliability problem.
- It builds statistically precise reliability analysis for FSM computations based on rigorous mathematical models.
- It provides a lightweight reliability analysis based on a two-state Markov model, which enables trading off the accuracy for faster analysis.
- By targeting a basic computation model, this work offers insights for the error tolerance and reliability analysis of a wider range of FSM-based applications.

In the following, we will first characterize the error tolerance in FSM computations, then formalize the reliability analysis of FSM computations in Section 3, followed by two reliability analysis schemes in Section 4 (full reliability analysis) and Section 5 (fast reliability analysis). After evaluating RelyFSM in Section 6, we will discuss the related work in Section 7 and conclude this work in Section 8.

2 ERROR TOLERANCE CHARACTERIZATION

In this section, we informally introduce the error types in FSM computations and discuss the fundamental causes of the error tolerance—*synchronization structures*—followed by a systematic

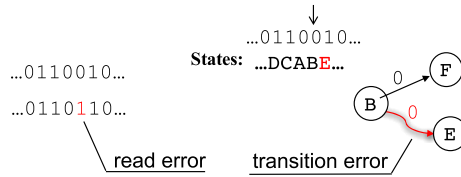


Fig. 4. Two basic types of errors

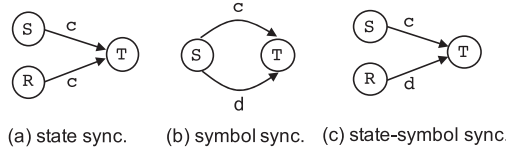


Fig. 5. Synchronization structures.

classification of various error tolerance cases. By revealing the essence of error tolerance, we show that error tolerance capabilities are prevalent in FSM computations.

2.1 Types of Errors

Informally, FSM computations consist of a sequence of state transitions, based on predefined transition rules. Each transition consists of two basic operations: (1) read an input symbol and (2) move to the next state. From the perspective of FSMs, the unreliable executions of the two operations form two basic sources of errors, as illustrated in Figure 4:

- *Read error*: In this case, an FSM reads an incorrect input symbol. As shown in Figure 4 (left), an FSM is supposed to read the symbol \emptyset but instead it reads 1.
- *Transition error*: After reading an input symbol, the FSM may transition to a state other than the one specified by the transition rule. As illustrated in Figure 4 (right), after reading a \emptyset , the FSM should transition to state F. However, due to an error, it moves to state E.

At the hardware level, the preceding FSM execution errors could stem from the use of unreliable arithmetic and logic units [12, 53] or/and the use of unreliable components in the memory hierarchy (e.g., unreliable registers, caches, or DRAM) for storing inputs and transition rules [47].

2.2 Synchronization Structures

Unlike general programs, FSM computations tend to exhibit high error tolerance capabilities. To understand the root causes of such error tolerance, we look into the structural diagrams of FSMs, like the one shown in Figure 1(c). We find that the error tolerance in FSM computations is enabled by three kinds of “structures” in the FSM structural diagrams, namely *state synchronization*, *symbol synchronization*, and *state-symbol synchronization*, as illustrated in Figure 5:

- *State synchronization*: In this case, two different states transition to the same state after reading the same symbol, as illustrated by the example in Figure 5(a).
- *Symbol synchronization*. In this structure, different input symbols from one source state point to the same target state, like the one in Figure 5(b).
- *State-symbol synchronization*: In the last case, there are two different states transitioning to the same state after reading two different symbols, as shown in Figure 5(c).

Together, we refer to the three structures as *synchronization structures*. It is interesting to note that the synchronization structures are so basic that almost all FSMs consist of some of them. In fact, the only exceptions are the trivial FSMs with a single-symbol alphabet and a state-chain

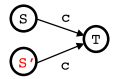
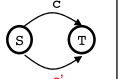
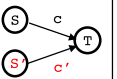
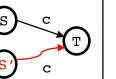
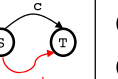
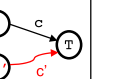
Cases		     					
		T1	T2	T3	T4	T5	T6
sync.	state sync.	✓			✓		
	symbol sync.		✓			✓	
	state-sym sync.			✓			✓
errors	in an err. state	✓		✓	✓		✓
	read error		✓	✓		✓	✓
	trans. error				✓	✓	✓
error cancellation				✓	✓	✓	✓

Fig. 6. Error tolerance types and corresponding causes/conditions.

structure. Consider the example FSM in Figure 1(c). It exhibits the symbol synchronization from state A to state B (i.e., A transitions to B on both \emptyset and 1) and state synchronization at state B, which owns three incoming edges labeled with \emptyset . Next, we show how synchronization structures enable the error tolerance in FSM computations in detail.

2.3 Qualitative Characterization of Error Tolerance

To systematically reveal the connections between error tolerance and synchronization structures, we categorize the error tolerance in FSM computations into six basic cases, denoted as **T1** through **T6** (see row 1 in Figure 6). The notations S' and c' indicate erroneous state and input symbol, respectively. The curved arrows represent erroneous transitions (more formal definitions are in Section 3). Note that this classification includes scenarios where multiple read and/or transition errors occur.

Considering **T1** first, its interpretation is that if the FSM was already in the error state S' rather than the correct state S , then the error gets tolerated after reading input symbol c , as both states transition to state T after reading c (i.e., state synchronization). Note that in this case, no error happens in the current transition. This is exactly the case mentioned earlier in motivation example (Figure 1(b)). Next, consider **T2**, where the FSM was in the current state S but the input symbol is read incorrectly. Fortunately, both the incorrect input and the correct input lead the FSM to the same next state T . As a result, the FSM remains in the correct state. The following cases **T3**, **T4**, and **T6** are similar to **T1** except they suffer from error(s) in the current transition. The case **T5** is similar to **T2** except it consists of both input and transition errors. Note that the last four cases (**T3**–**T6**) all involve some form of *error cancellation*—later error(s) cancels out the effect of the prior error(s). More detailed conditions and causes of different cases are listed in the following rows of Figure 6. One important thing worth to mention is that all the six basic classes of error tolerance require one synchronization structure (see rows 2–4).

Figure 7 shows a simple way to identify the type of errors based a two-layer decision tree. The first layer specifies if the FSM was already in the error state, and the second layer is concerned with the errors happening in the current transition. Note that when the FSM was in the correct state, it had no chance to tolerate a transition error, according to the definition of transition error.

3 FORMALIZATION

To rigorously analyze and verify its reliability, we formally define FSM and its execution semantics, including both the error-free (reliable) and potentially faulty (unreliable) executions. Based on the semantics, we further formalize the reliability analysis problem for FSM computations.

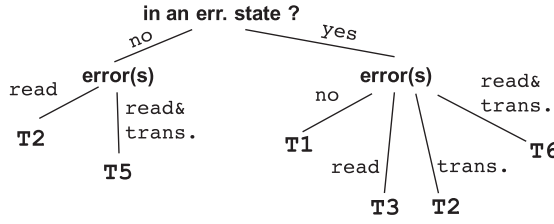


Fig. 7. A decision tree for error tolerance classification.

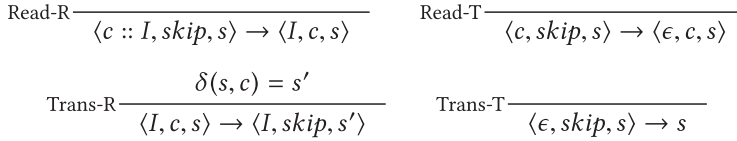


Fig. 8. Semantics for reliable FSM execution.

3.1 FSM Definition

An FSM is an abstract computation model formally defined as follows.

Definition 3.1. An FSM \mathcal{M} is a 5-tuple $(Q, \Sigma, \delta, s_0, F)$, where Q is a finite state set called *states*, Σ is a finite set called the *alphabet*, each element $c \in \Sigma$ is called a *symbol*, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, $s_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accept states.

Note that FSM has different variations. In this work, we assume that the FSM is deterministic, also known as *Deterministic Finite Automata* (DFA). For a given input I , a deterministic FSM executes from the initial state s_0 and consumes one symbol from the input string each time. Based on the read symbol c and the current state s , the FSM transitions to the next state according to the transition function $\delta(s, c)$. The FSM execution terminates when the whole input string has been consumed. However, under an unreliable computing environment, the FSM may exhibit different behaviors. To rigorously compare its behaviors in reliable and unreliable environments, we next formally define the semantics of reliable and unreliable FSM executions.

3.2 Semantics of Reliable FSM Execution

An *environment* of FSM execution is a 3-tuple of input, symbol, and state $\langle I, c, s \rangle$, meaning that the FSM currently is in state s and it has read input symbol c , with input sequence I to process. We use notation $c :: I$ to represent an input sequence starting with symbol c .

Figure 8 shows the small-step semantics for reliable FSM executions. In particular, Read-R defines the effect of a read operation, where the next symbol from the input is consumed; Read-T indicates the case where the last symbol in the input is consumed; Trans-R defines the effect of a transition, where the read symbol c is used and the state is updated from s to s' , under the precondition that $\delta(s, c) = s'$ exists; and Trans-T defines the termination of the FSM execution, where the input is empty and there is no read symbol to be available. Overall, an FSM execution consists of a sequence of interleaved read and trans steps.

With the small-step semantics, we further define the big-step semantics as the transitive closure of a sequence of small-step evaluations: $\langle I, s \rangle \Rightarrow s' \equiv \langle I, s \rangle \rightarrow \dots \rightarrow \langle skip, s' \rangle$, which captures the process of a (partial) FSM execution where the input I is fully consumed and the FSM finally ends at state s' after a series of small-step operations. Note that here, s could be any state in Q , including, but not necessarily limited to, the initial state s_0 .

$$\begin{array}{c}
\text{Read-C} \frac{c :: I \quad p = 1 - e_r}{\langle \text{skip}, s \rangle \xrightarrow{C,p} \langle c, s \rangle} \qquad \text{Read-F} \frac{c :: I \quad c_f \neq c \quad p = e_r \times P(c_f|c)}{\langle \text{skip}, s \rangle \xrightarrow{\langle F, c_f \rangle, p} \langle c_f, s \rangle} \\
\text{Trans-C} \frac{\delta(s, c) = s' \quad p = 1 - e_t}{\langle c, s \rangle \xrightarrow{C,p} \langle \text{skip}, s' \rangle} \qquad \text{Trans-F} \frac{s_f \neq \delta(s, c) \quad p = e_t \times P(s_f|s, c)}{\langle c, s \rangle \xrightarrow{\langle F, s_f \rangle, p} \langle \text{skip}, s_f \rangle}
\end{array}$$

Fig. 9. Semantics for unreliable FSM execution.

3.3 Semantics of Errors

According to its execution semantics, an FSM execution consists of two basic small steps: (1) read the next input symbol (*read*) and (2) transition to the next state (*trans*). The error(s) might happen in either step or both. Formally, we define two types of errors as follows²:

- **Read error:** Suppose that the next symbol at the beginning of the remaining input sequence is c (i.e., $c :: I$), then the FSM reads a symbol c_f . If $c_f \neq c$, we say that a read error has happened, where symbol c_f is called the *faulty symbol*. The probability that a read error occurs is denoted as e_r . The distribution of a faulty symbol is a mapping from the alphabet Σ to a probability vector, denoted as $\Sigma \rightarrow [P_0, P_1, \dots, P_{|\Sigma|-1}]$, where P_i is the probability that c_i is the faulty symbol. In addition, we have $\sum_i P_i = 1$ and $P_f = 0$.
- **Trans error:** Suppose that symbol c has just been consumed from the input and FSM is in state s (i.e., $\langle I, c, s \rangle$), then the FSM makes a transition, ending at state s_f . If $s_f \neq \delta(s, c)$, we say that a trans error has just occurred. State s_f is referred to as the *faulty state*. The probability that a trans error occurs is denoted as e_t . The distribution of a faulty state is a mapping from the state set Q to a probability vector, denoted as $Q \rightarrow [P_0, P_1, \dots, P_{|Q|-1}]$, where P_i is the probability that s_i is the faulty state. In addition, we have $\sum_i P_i = 1$ and $P_f = 0$.

Calculating e_r and e_t . In practice, the error probabilities of read and trans can be calculated as the joint (error) probabilities of machine instructions that implement the read and trans, following the conventional instruction-level reliability analysis [8, 39]. For example, assume that an FSM read is implemented with an add instruction followed by a mov instruction, then $e_r = e_{add} \cdot e_{mov}$, where e_{add} and e_{mov} are the error probabilities of add and mov, respectively. Essentially, there are two levels of error models: the conventional instruction-level error model and the FSM-level error model. The error rates of the latter depends on the error rates of the former.

3.4 Semantics of Unreliable FSM Execution

Based on the preceding error semantics, we next define the semantics for unreliable FSM execution, as summarized in Figure 9. A small-step evaluation relation (e.g., $\langle \text{skip}, s \rangle \rightarrow \langle c, s \rangle$) carries two labels. The first label θ , $\theta \in \{C, \langle F, c_f \rangle, \langle F, s_f \rangle\}$, denotes whether the evaluation is correct (C) or faulty (F), and if it is faulty, what the fault symbol or state (c_f or s_f) is. The second label p indicates the probability that a correct/faulty evaluation occurs. For example, the first rule Read-C says that if the next symbol is c , there is a probability of $1 - e_r$ that the evaluation is correct: $\langle \text{skip}, s \rangle \rightarrow \langle c, s \rangle$. Note that for faulty read rule Read-F, the probability of the faulty evaluation is calculated as the product between error probability of read e_r and the conditional probability $P(c_f|c)$. The latter denotes the probability that the faulty symbol is c_f , given that the next symbol is c . Similarly, the faulty trans rule Trans-F needs to take the conditional probability $P(s_f|s, c)$ into account,

²Note that if an error generates an undefined symbol or state, the error can be caught immediately by range checking.

which represents the probability that the faulty state is s_f , given that the prior state is s and the last consumed symbol is c . Together, the four rules in Figure 9 define the possible behaviors of an unreliable FSM execution at each small-step evaluation.

Based on the small-step semantics, we next introduce the *big-step trace semantics* and *big-step aggregate semantics* for unreliable FSM executions, following a similar style as those in recent work [4]. Basically, big-step trace semantics defines the whole unreliable FSM execution as a transitive closure of a sequence of small-step evaluations. The labels on the small-step evaluations together form a *trace*, denoted as τ , $\tau = \theta_1, \dots, \theta_n$. The probability of a trace p is the product of probabilities of all individual small-step evaluations—that is, $p = \prod p_i$. Thus, we have the following:

- *Big-step trace semantics:*

$$\langle I, s \rangle \xrightarrow{\tau, p} s' \equiv \langle I, s \rangle \xrightarrow{\theta_1, p_1} \dots \xrightarrow{\theta_n, p_n} \langle \text{skip}, s' \rangle, \text{ where } \tau = \theta_1, \dots, \theta_n \text{ and } p = \prod p_i.$$

In sum, $\langle I, s \rangle \xrightarrow{\tau, p} s'$ states that given input I and state s , an unreliable FSM execution has probability p to exactly follow a specific sequence of (correct/faulty) small-step evaluations.

By aggregating all unreliable execution traces (assuming the trace set is denoted as T) and their corresponding probabilities, we can further define the big-step aggregate semantics:

- *Big-step aggregate semantics:*

$$\langle I, s \rangle \xrightarrow{p} s', \text{ where } p = \sum_{\tau \in T} p_\tau \text{ such that } \langle I, s \rangle \xrightarrow{\tau, p_\tau} s'.$$

Intuitively, the preceding evaluation tells that given an input I and a state s , an unreliable FSM execution has a probability of p ending at state s' . Note that the probability p may include a number of alternative sequences of small-step evaluations that end at state s' , different from the probability in big-step trace semantics, which focuses on a specific sequence of small-step evaluations.

As we will show next, the preceding two big-step semantics form the theoretical foundations for defining the reliability of unreliable FSM executions.

3.5 Reliability of Unreliable FSM Execution

Depending on the strictness of correctness criteria, we introduce two basic types of reliability, namely *trace reliability* and *state reliability*.

Trace reliability. The correctness criterion for the trace reliability is strict—an unreliable FSM execution should behave exactly the same as the reliable FSM execution semantically (as defined in Section 3.2). Based on the big-step trace semantics, we define the *trace correctness* as follows.

Definition 3.2. For an unreliable FSM execution, $\langle I, s \rangle \xrightarrow{\tau, p} s'$, where the trace $\tau = \theta_1, \dots, \theta_n$. If $\theta_i = C$, $1 \leq i \leq n$, then the unreliable FSM execution satisfies the *trace correctness*.

Trace correctness requires every single small-step evaluation to be performed correctly during the whole FSM execution. Assume that the length of an input is $|I|$, then there would be $|I|$ read and trans operations, respectively, interleaved with each other during the FSM execution. Based on the definitions of big-step trace semantics and the trace probability ($p = \prod p_i$), we can infer the probability that an unreliable FSM execution meets the trace correctness—the *trace reliability*.

THEOREM 3.3. *The trace reliability of an unreliable FSM execution on input I is*

$$\mathcal{R}_\tau = ((1 - e_r) \times (1 - e_t))^{|I|}, \quad (1)$$

where e_r and e_t are the error probabilities of read and trans, and $|I|$ is the input length.

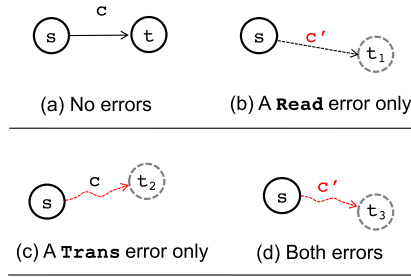


Fig. 10. Four possible cases in an unreliable transition.

As discussed in Section 2.3, errors occurred during the unreliable FSM execution may be tolerated due to a variety of reasons and may still end up in the correct state. The preceding trace reliability is too strict to accommodate such error tolerance. To capture the error tolerance in unreliable FSM executions, we next introduce the *state reliability*, which is more forgiving than trace reliability.

State reliability. The correctness for state reliability is solely based on the correctness of the ending state, regardless the correctness of small-step evaluations. This idea perfectly matches the big-step aggregate semantics defined in Section 3.4. Formally, we define the *state correctness* as follows, based on a pair of reliable and unreliable FSM executions.

Definition 3.4. For an unreliable FSM execution $\langle I, s \rangle \xrightarrow{\tau, p} s'$, if final state s' equals to the final state s'' in the corresponding reliable FSM execution $\langle I, s \rangle \Rightarrow s''$, then we say the unreliable FSM execution satisfies the *state correctness*.

Similar to trace reliability, we define the *state reliability* as the probability that an unreliable FSM execution satisfies the state correctness, denoted as \mathcal{R}_s . Thanks to the possible error tolerance in FSM executions, we can infer that $\mathcal{R}_\tau \leq \mathcal{R}_s$. Unlike trace reliability, state reliability is much more challenging to calculate, due to the complexities of error tolerance. In the following sections, we will mainly focus on the calculation of state reliability.

4 FULL RELIABILITY ANALYSIS

This section introduces a reliability analysis for unreliable FSM executions, with the goal to provide the precise answer to the state reliability problem raised in Section 3.5. This is achieved with a rigorous mathematical model that naturally combines the FSM with the probabilistic nature of unreliable computing. In the following, we will first examine the reliability for a single unreliable FSM transition, then discuss the reliability reasoning for an entire unreliable FSM execution.

4.1 Reliability of an Unreliable FSM Transition

First, we discuss how errors affect the behavior of a single FSM transition, based on which we then analyze how likely a single FSM transition tolerates the errors and still transitions to the correct state—that is, the reliability of a single FSM transition.

Behaviors of an unreliable transition. Consider an FSM transition in an unreliable computing environment. As discussed earlier, a read error and/or a trans error may occur. Depending on their occurrences, there are four possible correct/faulty transitions, as illustrated in Figure 10:

- (1) When no errors occurs, the FSM would follow its default transition $\delta(s, c) = t$ (Figure 10(a)).

Table 1. Notations and Probabilities of Possible Cases in an Unreliable Transition $\hat{\delta}$

Case	Notation	Probability
No error	$\hat{\delta}_{00}$	$P(\hat{\delta}_{00}) = \mathcal{R}(\text{read}) \cdot \mathcal{R}(\text{trans})$
Read error only	$\hat{\delta}_{10}$	$P(\hat{\delta}_{10}) = (1 - \mathcal{R}(\text{read})) \cdot \mathcal{R}(\text{trans})$
Trans error only	$\hat{\delta}_{01}$	$P(\hat{\delta}_{01}) = \mathcal{R}(\text{read}) \cdot (1 - \mathcal{R}(\text{trans}))$
Both errors	$\hat{\delta}_{11}$	$P(\hat{\delta}_{11}) = (1 - \mathcal{R}(\text{read})) \cdot (1 - \mathcal{R}(\text{trans}))$

- (2) When only a read error occurs, the FSM may transition to a different state, depending if the state t_1 in $\delta(s, c') = t_1$ is the same as t (Figure 10(b)).
- (3) When only a trans error occurs, the FSM must transition to a different state t_2 (i.e., $t_2 \neq t$) by the definition of a trans error (Figure 10(c)).
- (4) When both read and trans errors occur, the FSM would transition to a state t_3 such that $t_3 \neq \delta(s, c')$. Note that t_3 may equal to t if $\delta(s, c) \neq \delta(s, c')$ (Figure 10(d)).

For conciseness, we use $\hat{\delta}$ to represent an unreliable FSM transition and $\hat{\delta}_{xx}$, where $x \in \{0, 1\}$ to represent the four cases: the first x is for read error and the second one is for trans error, as shown in the second column of Table 1. The probabilities of the four cases are easy to calculate, as shown in the third column of Table 1.

According to the reliability definitions (see Section 3.5), we can find that the trace reliability of a single unreliable FSM transition is $\mathcal{R}_\tau = P(\hat{\delta}_{00})$. However, to find out the result reliability \mathcal{R}_{result} , we need to reason about the possibility that state s_i successfully transitions to the correct next state s_j after reading the symbol c in an unreliable transition, denoted as $P(\hat{\delta}(s_i, c) = s_j)$. For example, we need to find out the possible states t_1 , t_2 , and t_3 in Figure 10 and their probabilities.

Reliability reasoning for an unreliable transition. At high level, depending on the outcome of an unreliable transition, we can break down the reasoning into two situations. For a reliable transition $\delta(s_i, c) = s^*$, the two possible situations of an unreliable transition $\hat{\delta}(s_i, c) = s_j$ are

- $$\begin{cases} \text{Situation I :} & \text{the target state } s_j \text{ is the correct state (i.e., } s_j = s^*); \\ \text{Situation II :} & \text{the target state } s_j \text{ is a wrong one (i.e., } s_j \neq s^*). \end{cases}$$

We next analyze the two situations one by one. To simplify the formal representations, we use *transition matrix* $M_c(s_i, s_j)$ to represent all valid transitions under input symbol c , $c \in \Sigma$.

$$\begin{aligned} M_c : Q \times Q &\rightarrow \{0, 1\} \\ \text{subject to } M_c(s_i, s_j) &= 1 \text{ if and only if } \delta(s_i, c) = s_j \end{aligned} \quad (2)$$

Basically, M_c is a Boolean matrix. For symbol c , if there exists a valid transition from state s_i to state s_j , then $M_c(s_i, s_j) = 1$; otherwise, $M_c(s_i, s_j) = 0$.

Situation I. To calculate the probability that an unreliable transition $\hat{\delta}(s_i, c)$ moves to the correct next state s^* —that is, $P(\hat{\delta}(s_i, c) = s^*)$ —consider the four cases in Table 1:

- (1) When no error happens (δ_{00}), the FSM must transition to the correct state s^* , so the conditional probability that s_i transitions to s^* is $P(\hat{\delta}(s_i, c) = s^* | \delta_{00}) = 1$.
- (2) When only a read error happens (δ_{10}), the FSM reads a faulty symbol c_f . Assume that c_f has a probability of $P(c_k | c)$ to be symbol c_k . If c_f also leads state s_i to state s^* (i.e., $\delta(s_i, c_f)$

$= s^*$), the FSM still transitions to the correct state. Hence, we have

$$P(\hat{\delta}(s_i, c) = s^* | \delta_{10}) = \sum_k P(c_k | c) \cdot M_{c_k}(s_i, s^*). \quad (3)$$

- (3) When only the trans error happens (δ_{01}), according to the definition of a trans error, the FSM cannot transition to s^* , and therefore $P(\hat{\delta}(s_i, c) = s^* | \delta_{01}) = 0$.
- (4) When both errors happen (δ_{11}), the FSM may transition to the correct state only when the read error causes the FSM to transition to a wrong state s_j ($s_j \neq s^*$) and the following trans error accidentally moves the FSM to the correct state s^* , with a probability of $P(s^* | s_j)$. Hence,

$$P(\hat{\delta}(s_i, c) = s^* | \delta_{11}) = \sum_{s_j \neq s^*} P(c_k | c) \cdot M_{c_k}(s_i, s_j) \cdot P(s^* | s_j). \quad (4)$$

Based on the probabilities in Table 1, we can put the preceding four cases together to calculate the probability that the unreliable transition moves the FSM to the correct state—the reliability of a single unreliable transition, summarized by the following theorem.

THEOREM 4.1. *The reliability of an unreliable transition $\mathcal{R}(\hat{\delta}(s_i, c) = s^*)$ is given as follows:*

$$\begin{aligned} \mathcal{R}(\hat{\delta}(s_i, c) = s^*) &= P(\hat{\delta}_{00}) + P(\hat{\delta}_{01}) \cdot \sum_k P(c_k | c) \cdot M_{c_k}(s_i, s^*) \\ &+ P(\hat{\delta}_{11}) \cdot \sum_{s_j \neq s^*} P(c_k | c) \cdot M_{c_k}(s_i, s_j) \cdot P(s^* | s_j), \end{aligned} \quad (5)$$

where $s^* = \delta(s_i, c)$.

Theorem 4.1 implies that even for a single unreliable transition, it is possible that $\mathcal{R}(\hat{\delta}(s_i, c) = s^*)$ is greater than \mathcal{R}_τ , depending on the transition structure of the FSM.

Situation II. Next, we discuss the probability that an unreliable transition $\hat{\delta}(s_i, c)$ moves to a wrong state s_j —that is, $P(\hat{\delta}(s_i, c) = s_j)$, where $s_j \neq s^*$. Although it does not provide the reliability information directly, it is needed for precise reliability propagation, as we will show shortly in Section 4.2. Similarly to Situation I, let us consider the four cases in Table 1:

- (1) When no error happens (δ_{00}), the FSM transitions to the correct state s^* , so the conditional probability of entering a wrong state s_j is zero—that is, $P(\hat{\delta}(s_i, c) = s_j | \delta_{00}) = 0$.
- (2) When only a read error happens (δ_{10}), the FSM reads a faulty symbol c_f with a probability of $P(c_k | c)$ to be symbol c_k ($c_k \neq c$). If c_f leads state s_i to state s_j (i.e., $\delta(s_i, c_f) = s_j$), the FSM would end in the wrong state s_j . Hence, we have

$$P(\hat{\delta}(s_i, c) = s_j | \delta_{10}) = \sum_k P(c_k | c) \cdot M_{c_k}(s_i, s_j). \quad (6)$$

- (3) When only the trans error happens (δ_{01}), assume that the FSM has a probability of $P(s_j | s_i)$ transitioning to s_j , then we have $P(\hat{\delta}(s_i, c) = s_j | \delta_{01}) = P(s_j | s_i)$.
- (4) When both errors happen (δ_{11}), the FSM may transition to the wrong state s_j only when the read error causes the FSM to transition to a state s'_j ($s'_j \neq s_j$) and the following trans error accidentally moves the FSM to state s_j , with a probability of $P(s_j | s'_j)$. Hence, we have

$$P(\hat{\delta}(s_i, c) = s_j | \delta_{11}) = \sum_{s'_j \neq s_j} P(c_k | c) \cdot M_{c_k}(s_i, s'_j) \cdot P(s_j | s'_j). \quad (7)$$

Putting the four cases together, we have the probability that, given the current state s_i and symbol c , an unreliable transition would end at (incorrect) state s_j .

$$P(\hat{\delta}(s_i, c) = s_j) = P(\hat{\delta}_{10}) \cdot \sum_k P(c_k | c) \cdot M_{c_k}(s_i, s_j) + P(\hat{\delta}_{01}) \cdot P(s_i | s'_j) \\ + P(\hat{\delta}_{11}) \cdot \sum_{s'_j \neq s_j} P(c_k | c) \cdot M_{c_k}(s_i, s'_j) \cdot P(s_j | s'_j), \quad (8)$$

where $s_j \neq s^*$. So far, we have derived the probabilities that an transition ends in the correct state s^* and the incorrect state s_j , respectively. Next, we show how to extend the reliability reasoning from one unreliable transition to an entire FSM execution.

4.2 Reliability of an Entire FSM Execution

The key to answer the reliability of an entire FSM execution is to understand how errors are propagated transition by transition probabilistically. In fact, such a probability propagation can be modeled as a series of vector-matrix multiplications.

Probabilistic transition matrix. As mentioned earlier, for input symbol c ($c \in \Sigma$), the (reliable) FSM transitions can be represented as a binary matrix, $M_c(s_i, s_j)$. The potential errors, in fact, make the transitions probabilistic. In other words, the transition matrix for input symbol c becomes probabilistic, which we refer to as the *probabilistic transition matrix* for symbol c , denoted as \hat{M}_c .

$$\hat{M}_c = \begin{bmatrix} P(\hat{\delta}(s_1, c) = s_1) & P(\hat{\delta}(s_1, c) = s_2) & \dots & P(\hat{\delta}(s_1, c) = s_n) \\ P(\hat{\delta}(s_2, c) = s_1) & P(\hat{\delta}(s_2, c) = s_2) & \dots & P(\hat{\delta}(s_2, c) = s_n) \\ \vdots & \vdots & \ddots & \vdots \\ P(\hat{\delta}(s_n, c) = s_1) & P(\hat{\delta}(s_n, c) = s_2) & \dots & P(\hat{\delta}(s_n, c) = s_n) \end{bmatrix}, \quad (9)$$

where $P(\hat{\delta}(s_i, c) = s_j)$ represents the probability that an unreliable transition from state s_i to state s_j after reading input c . Note that this s_j could be the correct target state or any incorrect one, which correspond to the two situations analyzed in Section 4.1. Therefore, we have already derived their probabilities with Equation (5) and Equation (8), respectively.

With the probabilistic transition matrix, we can derive the state distribution π_i after consuming each input symbol c_i in an input sequence $c_1 c_2 \dots c_i \dots c_L$ with vector-matrix multiplications. Suppose that the initial state distribution is $\pi_0 = [00 \dots 1 \dots 0]$, where 1 corresponds to the initial state s_0 , then the new distribution after reading input symbol c would be $\pi_0 \cdot \hat{M}_c$. In general, we can calculate the final state distribution π_L iteratively,

$$\pi_L = \pi_0 \cdot \hat{M}_{c_1} \cdot \hat{M}_{c_2} \dots \hat{M}_{c_L}, \quad 1 \leq L, \quad (10)$$

where \hat{M}_{c_i} is the probabilistic transition matrix for c_i . The time complexity of the calculation is $O(L \cdot N^2)$, where L is input length and N is the number of states.

Take the FSM in Figure 1(c) as an example. Given error rates e_r and e_t , we can construct the probabilistic transition matrices for symbols 0 and 1 based on Equations (5) and (8). Then, for a given unreliable FSM execution on input 0010, the state distribution can be computed as follows:

$$\begin{bmatrix} \pi_4(A) \\ \pi_4(B) \\ \pi_4(C) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} \dots \\ \dots \\ \dots \end{bmatrix}_0 \cdot \begin{bmatrix} \dots \\ \dots \\ \dots \end{bmatrix}_0 \cdot \begin{bmatrix} \dots \\ \dots \\ \dots \end{bmatrix}_1 \cdot \begin{bmatrix} \dots \\ \dots \\ \dots \end{bmatrix}_0, \quad (11)$$

where $\pi_4(i)$ is the probability of ending at state i after the unreliable FSM execution over 0010.

Two things are worth noticing in the calculation of π . First, because matrix multiplications are associative, the calculation can be easily parallelized in a prefix sum manner. However, this will increase the time complexity to $O(N^3 \cdot \log(L))$.³ Second, the order of the input symbols matters to the state distribution since matrix multiplication is not commutative in general. For example, the state distribution after reading 1010 could be different from that of 0010. This also reflects the fact that π provides a per-input state distribution.

With the state distribution π_L , to find out the reliability of the ending state, all we only need to know is which state the FSM is supposed to be after processing the input. Consider the example in Figure 1(c). Suppose the FSM starts from state A, after processing inputs 0010 in a reliable execution, it ends at state B, then the reliability of an unreliable execution on the same input should be the probability of ending at state B—that is, $\pi_4(B)$. In general, we have the following theorem.

THEOREM 4.2. *Given an FSM \mathcal{M} and its unreliable execution with error rates e_r and e_t on input I of length L , suppose that the final state distribution after processing I is π_L and the ending state of its reliable execution is s_i^* , then the reliability of the unreliable execution is $\pi_L(i^*)$.*

We refer to the preceding analysis as *full reliability analysis*. Full reliability analysis precisely reasons about the unreliable behaviors of an unreliable FSM execution step by step, thus producing the “ground truth” of reliability, for which conventional fault-injection-based approaches never achieve.

Although full reliability analysis provides precise reliability results, its time complexity $O(L \cdot N^2)$ is significantly higher than FSM execution complexity $O(L)$. Such a gap prevents it from being applied in some online analysis scenarios. To address this, we introduce a lightweight reliability analysis approach, which we refer to as *fast reliability analysis*.

5 FAST RELIABILITY ANALYSIS

The goal of fast reliability analysis is to provide relatively accurate reliability results, yet run dramatically faster than full reliability analysis. To achieve this goal, we put efforts along two directions of optimizations: (i) partially characterize the error tolerance offline and (ii) simplify the probability model used for reasoning. Following the two directions, we propose a *dual-state Markov model* based on *offline error tolerance characterization* for more efficient reliability analysis. We next elaborate on its basic ideas.

5.1 Offline Error Tolerance Characterization

To reduce the cost of (online) analysis, we characterize the error tolerance capability of FSMs offline. The characterization is based on the error tolerance characterizations in Section 2.

Error tolerance parameters. Recall that in Section 2 we revealed six basic cases of error tolerance (T1–T6 in Figure 6). Here, we quantify the error tolerance by measuring the probabilities that the six error tolerance cases happen in unreliable executions. For convenience, we denote these probabilities as P_i ($1 \leq i \leq 6$), referred to as *error tolerance parameters*.

Also discussed in Section 2, the error tolerance cases are due to three basic synchronization structures. Suppose that the likelihoods the three synchronization structures are encountered during an FSM execution are α (state sync.), β (symbol sync.), and γ (state-symbol sync.). Then, for a given pair of read and transition error rates, e_r and e_t , we can calculate the error tolerance parameters, as shown in Figure 11. Note that $P(\delta_{00})$, $P(\delta_{10})$, $P(\delta_{01})$, and $P(\delta_{11})$ are the transition probabilities from Table 1. Next, we explain how α , β , and γ could be calculated.

Here, we discuss two approaches, based on static analysis and dynamic profiling, respectively:

³Assume a basic matrix multiplication algorithm with $O(N^3)$ complexity.

$$P_1 = P(\delta_{00}) \cdot \alpha \quad (12)$$

$$P_2 = P(\delta_{10}) \cdot \beta \quad (13)$$

$$P_3 = P(\delta_{10}) \cdot \gamma \quad (14)$$

$$P_4 = P(\delta_{01}) \cdot (1 - \beta)/(N - 1) \quad (15)$$

$$P_5 = P(\delta_{11}) \cdot (1 - \alpha)/(N - 1) \quad (16)$$

$$P_6 = P(\delta_{11}) \cdot (1 - \gamma)/(N - 1) \quad (17)$$

Fig. 11. Calculation of error tolerance parameters.

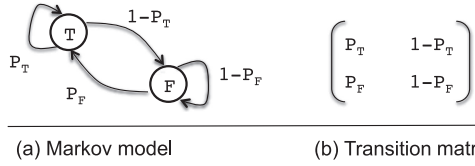


Fig. 12. Dual-state Markov model.

- *Static analysis*: In this method, we analyze the given FSM statically without any inputs. Take α as an example. To statically reason about the probability that a wrong input symbol can lead the FSM to the correct state, one can examine each state in the FSM, and check each pair of symbols (c, c') , where c is assumed as the correct symbol while c' is considered as the wrong one. By calculating the frequency ratio that a pair of symbols lead the state to the same next state, α can be computed. In similar ways, β and γ can be computed as well. Static analysis considers all possible inputs—that is, all symbol combinations of length L (i.e., Σ^L)—and hence it produces accurate values from a statistical point of view.
- *Dynamic profiling*: Alternatively, a more straightforward method is profiling these parameters with either offline training inputs or a small portion of the actual (testing) input. When using an actual testing input, it involves in a tradeoff between profiling accuracy and profiling cost.

For simplicity and efficiency, we use offline profiling when the error tolerance parameters are less input insensitive and online profiling otherwise. The sensitivity can be simply tested offline.

5.2 Dual-State Markov Model

To improve the efficiency of reliability analysis, our idea is to abstract the level of analysis from *concrete FSM states* to two abstract states: *correct state* and *incorrect state*. In this way, the probabilistic transition model in full reliability analysis is streamlined to a lightweight *dual-state Markov model*. Figure 12 illustrates this new model.

In this model, we represent the correct state as T and the incorrect state as F . In this case, it only needs two parameters to represent all of the transition probabilities, denoted as P_T and P_F . The former is the probability that an FSM stays in a correct state, whereas the latter is the probability that an FSM gets back to the correct state from a wrong state. Based on the error tolerance parameters from the offline error tolerance characterization (Section 5.1), P_T and P_F can be calculated as follows:

$$\begin{aligned} P_T &= P(\delta_{00}) + P_2 + P_5 \\ P_F &= P_1 + P_3 + P_4 + P_6. \end{aligned} \quad (18)$$

Note that the reliability after the n -th transition (denoted as $R(n)$) depends on the reliability after the $n - 1$ -th transition (denoted as $R(n - 1)$). This allows us to build a recursive function to

calculate the reliability. Obviously, $1 - R(n)$ is the probability of being in a wrong state after the n -th transition. According to the definitions of P_T and P_F , we have

$$R(n) = \begin{cases} R(n-1) \cdot P_T + (1 - R(n-1)) \cdot P_F & n > 0 \\ 1 & n = 0. \end{cases} \quad (19)$$

After simplification, we can get the following closed form:

$$R(n) = (P_T - P_F)^n + \frac{P_F(1 - (P_T - P_F)^n)}{1 - (P_T - P_F)} \quad (20)$$

This formula tells the reliability of the state after the FSM finishes the n -th transition. Note that since $|P_T - P_F| < 1$, we can calculate the limit of the reliability as follows:

$$\lim_{n \rightarrow \infty} R(n) = P_F / (1 - P_T + P_F). \quad (21)$$

This means that, statistically speaking, the state reliability tends to converge to a lower bound as the FSM executes, a phenomenon also observed in real-world FSM computations (see Section 6).

In sum, with the offline error tolerance characterization and state abstraction, we provide a new reliability analysis solution. The new analysis leverages a closed-form equation to calculate the reliability efficiently and also provides a statistical lower bound for the reliability.

6 EVALUATION

In this section, we first evaluate the proposed reliability analyses in terms of accuracy and cost, then demonstrate their uses with a case study in approximate searching of biological patterns.

6.1 Methodology

We first validate the correctness of full reliability analysis by comparing it with extensive *error injection-based simulations*, which mimic unreliable FSM executions a huge number of times (i.e., sampling) to collect the reliability statistically. For each simulation of unreliable FSM execution, we inject errors randomly (but not exhaustively) into the input and transitions based on the specified read and transition error rates (e_r and e_t), respectively. Note that since the proposed reliability analysis is at the FSM level, this evaluation does not include lower-level error injections, which could be conducted independently for collecting the FSM read and transition error rates. In the following, we treat the two error rates as the parameters to our analysis model. After the validation, we use full reliability analysis as the baseline to evaluate the accuracy and efficiency of fast reliability analysis. Note that the error injection-based approach is straightforward to implement but requires a high sampling rate to achieve high precision. For this reason, the evaluation also examines the efficiencies of the different reliability analysis approaches.

Benchmarks. We use the FSM benchmark suit from recent FSM studies [56, 57] and update it with FSMs from biological searching applications. Table 2 lists their basic information, including description, number of states, and symbols. They are drawn from a variety of areas, including *text mining*, *data decompression*, *pattern matching*, *bioinformatics*, and *mathematics*. Together, they cover a spectrum of complexities. The number of states ranges from several to thousands. For space limits, we only show the results of representative benchmarks when the results of others are similar.

6.2 Correctness of Full Reliability Analysis

As explained in Section 4, full reliability analysis is based on rigorous mathematical models and provides the ground truth of reliability. To validate this, we compare its results with those from extensive error injection-based simulations. The latter extensively simulates unreliable execution

Table 2. Benchmarks

Name	Description	#States	#Accept	Symbols
div7	Unary Divisibility	7	1	Binary
evenodd	Even Odd Testing	4	1	{abcd}
commadot	Text Searching	130	7	ASCII
likeapple	Text Searching	495	1	ASCII
huff	Huffman Decoding	511	256	ASCII
dna1	DNA Motif Search	371	76	{ATGC}
dna2	DNA Motif Search	2,871	583	{ATGC}
dna3	DNA Motif Search	40	5	{ATGC}
protn1	Protein Motif Search	68	6	20 Letters
protn2	Protein Motif Search	280	14	20 Letters
protn3	Protein Motif Search	831	48	20 Letters

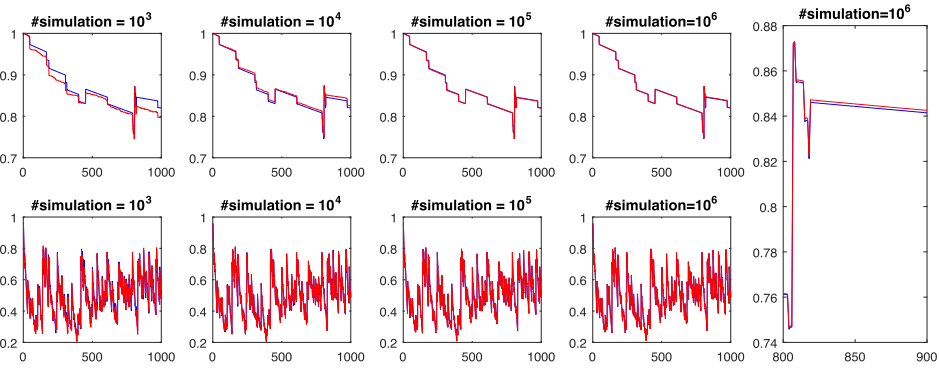


Fig. 13. Validation of full reliability analysis. The two rows are for *commadot* and *huff*, respectively. (x-axis: transitions, y-axis: reliability, blue lines: error injection-based simulation, red lines: full reliability analysis, and error rates are $e_r = e_t = 0.02$.)

under the given error rates (i.e., e_r and e_t). Figure 13 reports the reliability of the first 1,000 transitions for two benchmarks: *commadot* and *huff*. Results from other benchmarks are similar. The error rates are artificially set to high values ($e_r = e_t = 0.02$) to magnify the variations of reliability.

The results clearly show that the reliability from simulations tend to converge to the reliability from full reliability analysis; the differences between the two reduce as the amount of simulations increases. Note that the reliability curve of *huff* (on the right) varies quickly due to its complex transition structure. Even for this challenging case, we find full reliability analysis still perfectly captures its reliability variation. However, the results also demonstrate the key advantage of full reliability analysis—even the reliability resulted from 1M simulations of error-injected executions still exhibits noticeable discrepancy, as shown in the right-most subfigure in Figure 13. Moreover, the discrepancy often increases as more transitions are performed.

Discussion. Even though the preceding results confirm the correctness of full reliability analysis in theory, there could be some other factors affecting the actual reliability in practice. First, as full reliability analysis takes the read and trans error rates as inputs, the precisions of these error rates directly affect the reliability results. However, this impact can be statistically minimized with more intensive profiling (i.e., reducing the error margin). Second, in some application scenarios, there

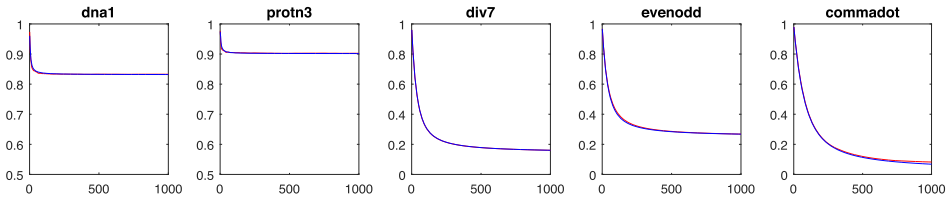


Fig. 14. Accuracy of fast reliability analysis. (x-axis: transitions, y-axis: accumulated reliability, blue lines: full reliability analysis of five inputs, red lines: fast reliability analysis, and $e_r = e_t = 0.02$.)

Table 3. Costs of Different Methods for Computing Reliability

Method	<i>div7</i> (7 states)			<i>protn1</i> (68 states)		
	1K	2K	4K	1K	2K	4K
Fast reliability analysis	0.291 ms	0.597 ms	1.21 ms	0.299 ms	0.576 ms	1.16 ms
Full reliability analysis	1.12 ms	2.37 ms	4.39 ms	94.7 ms	193 ms	378 ms
Error injection based	9.8 s	17.7 s	36.9 s	8.56 s	16.5 s	33.0 s

could be masking effects beyond FSM error tolerance, which the full reliability analysis does not take into account. For example, when the analysis is used for reasoning the reliability of sequential circuits, there might be cases in which the FSM enters a wrong state but the primary output remains correct. In such uses scenarios, our analysis provides a lower bound of the actual reliability.

6.3 Accuracy and Efficiency of Fast Reliability Analysis

We examine the accuracy and efficiency of fast reliability analysis based on full reliability analysis. The error tolerance parameters used in fast reliability analysis were collected using *dynamic profiling* (see Section 5.1), which yields higher-quality parameters overall, thanks to its input sensitivity. As fast reliability analysis abstracts state identities, we compare its accumulated reliability with that of full reliability analysis. The accumulated reliability is the averaged reliability among all transitions that have happened so far. As discussed in Section 5, the rationale is that many FSM applications care more about the overall reliability instead of the reliability of individual transitions.

Figure 14 shows the results of accumulated reliability on five representative benchmarks (in terms of reliability trends): *dna1*, *protn3*, *div7*, *evenodd*, and *commadot*. Although the benchmarks exhibit different reliability trends when reacting to the same error rates ($e_r = e_t = 0.02$), fast reliability analysis precisely captures the trends, with less than 1% discrepancy in the given setting. The results confirm the accuracy of fast reliability analysis and its potential for practical uses. Later in this section, we will show its uses for estimating the approximate pattern searching accuracies. Note that the precision loss of fast reliability analysis implies that the confidence of the reliability is reduced. When the confidence is of the highest preference, a full reliability analysis would be more preferred at the cost of a longer analysis time.

As to the efficiency, Table 3 summarizes the time costs of different methods for finding the reliability. For error injection–based simulation, the number of simulations is set to 100K. Both full reliability analysis and simulation show linear increases of costs as the input length grows, which is expected according to the complexities, $O(L \cdot N^2)$ and $O(L \cdot \#Simulation)$, discussed in Section 4. However, fast reliability analysis shows significant less cost than the other two approaches, thanks to its use of a closed-form equation for reliability calculation.

Table 4. Protein Motifs (x Represents Any Valid Symbol)

<i>protn1</i>	Description: <i>IQ calmodulin-binding motif</i> [FILV]-Q-x(3)-[RK]-G-x(3)-[RK]-x(2)-[FILVWY]
<i>protn2</i>	Description: <i>Hemopexin domain signature</i> [LIFAT]-[IL]-x(2)-W-x(2,3)-[PE]-x-[VF]- [LIVMFY]-[DENQS]-[STA]-[AV]-[LIVMFY]
<i>protn3</i>	Description: <i>P-type “Trefoil” domain signature</i> [KRH]-x(2)-C-x-[FYPSTV]-x(3,4)-[ST]- x(3)-C-x(4)-C-C-[FYWH]

6.4 Case Study: Approximate Searching in Biological Databases

To demonstrate the uses of the proposed reliability analysis, we conduct a case study on a specific yet important application: *pattern searching in biological databases*. In addition to the reliability results, we also roughly estimate the potential energy saving by adopting a disciplined approximate computing scheme. Next, we first motivate this case study and introduce the benchmarks.

Motivation. Due to the massive data sizes and the errors originally introduced by biological sequencing devices [54], *relaxed searching* has been proposed [5]. Here, “relaxing” means the results are allowed to miss some hits (i.e., false negatives) and contain incorrect hits (i.e., false positives). The pattern to search (in regular expression formats) can be converted to an FSM through the standard DFA construction algorithm, then the searching becomes an FSM execution, where each time the FSM enters an accept state, a hit is reported. By introducing errors at the FSM transition level, the FSM may not enter an accept state as expected or enter an accept state when it is not expected. Therefore, it is important to understand the reliability of the searching FSMs.

To introduce the read and trans errors, we consider the use of an approximate computing platform proposed by recent work: EnerJ [46]. In brief, EnerJ is based on a hybrid hardware model equipped with both approximate and reliable registers, data caches, memory regions, and functional units (similar platforms have been used by other work [4, 36]). The platform offers an approximation-aware ISA extension that allows application programs to distinguish between precise data and approximate data, as well as between precise computations and approximate computations. With these supports, we can limit the errors within the scope of the FSM executions. More specifically, there are three levels of approximation settings. Each setting corresponds to a different set of error rates. We refer to the three levels as *low*, *medium*, and *high*. By decomposing the FSM read and trans operations down to the instruction level, we can infer their error rates (e_r and e_t) with instruction-level analysis (see Section 3.3). More details about the approximation platform can be found in prior work [4, 36, 46].

Benchmarks. A search in biological database is to find a pattern that may exhibit biological significance, such as motifs. For example, a DNA motif is a short pattern of nucleic acid, whereas a protein motif is a pattern of amino acids. It is common to represent protein patterns with regular expressions. Table 4 lists three benchmark protein motifs randomly selected from the popular protein database PROSITE [14]. As to DNA motifs, they are more commonly represented with Hamming distances. In our benchmarks, *dna1* is a DNA motif for pattern ATCGGTCC(8, 3), which means three of eight symbols can be different from the specified ones. *dna2* and *dna3* are two other DNA motifs for patterns TCGAGGACCA(10, 4) and AGGGTAAAA(8, 1).

Both protein and DNA motifs can be implemented with FSMs. The basic statistics of the motif FSMs are shown in Table 2. An execution of a motif search returns all hits in the database.

Table 5. State Reliability in Relaxed Queries on Biological Sequences
(Input Sizes: 2,778,548,305 for DNA and 1,471,906,839 for Protein)

Benchmark	Err. Rate	#Err. States (actual)	#Err. States (est.)
dna1	Low	12,753	12,747
	Med	144,939	144,590
	High	14,382,834	14,369,394
dna2	Low	16,774	17,227
	Med	183,191	185,085
	High	18,438,646	18,569,717
dna3	Low	5,446	5,520
	Med	61,034	62,531
	High	6,007,384	6,133,971
protn1	Low	716	723
	Med	11,528	11,423
	High	1,165,197	1,191,223
protn2	Low	2,342	2,475
	Med	29,628	30,712
	High	2,936,596	3,057,013
protn3	Low	1,157	1,187
	Med	18,853	19,063
	High	1,870,571	1,884,260

Reliability of approximate searching. Table 5 reports the reliability results of approximated searches on a testing database with 2-GB DNA sequences and 1.5-GB protein sequences. The last two columns show the actual number of erroneous states in an unreliable search and the estimated number of erroneous states calculated by fast reliability analysis, respectively. A naive way to use fast reliability analysis here is accumulating the state reliability at every position of the input sequence, whereas a more efficient approach could leverage the convergence of state reliability, as fast reliability captures, to eliminate repeated calculation when the reliability has converged (according to some precision threshold). As the results shown, the calculated values estimate the actual numbers with high precision. Note that after an unreliability platform is deployed, there is no way to know the actual number of correct states in an unreliable execution. The proposed fast reliability provides a lightweight yet accurate estimation to help users understand the reliability of the search. Also note that the state reliability reported here is an “averaged” reliability of both accept and non-accept states. If users would like to distinguish between the two cases, some extensions to the fast reliability analysis would be needed or alternatively the users could use the full reliability analysis to achieve the same purpose (but at a higher cost).

Energy saving estimation. To estimate the potential energy saving, we adopt the energy model from a recent approximate computing framework [46], and collect the instruction-level parameters, such as the numbers of memory and arithmetic operations, to estimate the energy savings of the three approximation levels.⁴ Based on the instructions (produced by GCC 4.9.3 on a 64-bit Linux machine) of FSM execution, we find about 30% arithmetic operations, 50% SRAM operations, and about 20% DRAM operations. Based on the energy models, the estimated energy savings for the three levels of approximation are 20%, 23%, and 29%, respectively.

⁴More details can be found in Sampson et al. [46].

7 RELATED WORK

Fault tolerance and resilience are among the primary goals for building dependable computing systems. There is a rich body of research in developing methodologies for measuring the reliability at different levels of the computing stack, including, but not limited to, circuit-level reliability [7, 28, 48], architecture-level reliability [10, 16, 37, 50], instruction-level reliability [8, 39], operating system-level reliability [19], and application-level reliability [15, 21, 30–32, 43, 55]. However, there is limited work for studying the reliability at the computation model level, which is the focus of this work. There is early work on error analysis for finite automata [9], which discusses finite and infinite errors from the theoretical perspective. It does not characterize the causes of the error tolerance in detail and offers no quantitative analysis.

There is a series of studies on the quantitative reliability analysis of general programs. One of the early studies [50] defines the program vulnerability factor (PVF) to capture the architecture-level fault masking inherited in a program. Note that it captures both the crash-causing errors and silent-data corruption (SDC) errors. ePVF [15] improves PVF in that it focuses on the SDCs. However, it still includes benign faults that are tolerable by the program executions, due to its limitations in error propagation models. TRIDENT [31] and vTRIDENT [30] address the limitations by expanding the error propagation to control flows and the memory, yielding more accurate reliability estimations. However, this work shows that the preceding instruction-level error propagation methods cannot capture the reliability of FSM computations accurately, due to their inherent error tolerance (see Section 2). We address this gap with a state-level reliability analysis.

Approximate computing has emerged in the past few years [20], ranging from approximate hardware design [13, 35, 47] to the supports of architecture [12], programming languages [4, 36, 46], and even databases [22]. For example, approximate storage can be designed either using multi-level phase-change memory cells and failed blocks [47] or based on spintronic memories [42]. These efforts together build the infrastructure for approximate computing and call for the exploitation of more approximate applications. A core problem to this emerging domain is the reliability analysis [4, 36, 46]. A few recent studies provide programming language supports for reliability reasoning. For example, Rely [4] and Chisel [36] provide static reliability analysis to verify program reliability against predefined requirements. EnerJ [46] extends traditional assertions to probabilistic ones to support reliability analysis for approximate computing. Similarly to the work in fault tolerance studies, they propagate reliability at the instruction/statement level thus is unaware of the state-level error tolerance in FSM computations.

As a basic computation model, FSM has been well studied in theory, including the property of state synchronization [23, 25]. For practical uses, there is a series of efforts in parallelizing FSM executions [24, 38, 40, 41, 56, 57], most of which exploit the state synchronization in FSM computations. However, to the best of our knowledge, no prior work systematically examined the state synchronization properties for analyzing the reliability of FSM computations.

8 CONCLUSION

This work introduces RelyFSM, which to our best knowledge is the first quantitative reliability analysis for FSM computations. It provides a systematic way to characterize their intrinsic error tolerance properties, including the findings of three synchronization structures, the essential causes of the error tolerance in FSM computations, and the six basic classes of error tolerance cases. Based on the error tolerance characterization, this work provides both a rigorous full reliability analysis and a lightweight reliability analysis. The former can find the theoretical ground truth by simultaneously reasoning about the likelihood of each concrete state, whereas the latter trades off model accuracy for faster analysis. Experiments confirm the correctness and efficiency of the

proposed analyses and demonstrate the potential of adopting disciplined approximate computing for FSM computations.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their suggestions.

REFERENCES

- [1] Kevin Angstadt, Arun Subramaniyan, Elaheh Sadredini, Reza Rahimi, Kevin Skadron, Westley Weimer, and Reetuparna Das. 2018. ASPEN: A scalable In-SRAM architecture for pushdown automata. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*.
- [2] Robert C. Baumann. 2005. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability* 5, 3 (2005), 305–316.
- [3] Stephen D. Brown. 2007. *Fundamentals of Digital Logic with Verilog Design*. Tata McGraw-Hill Education.
- [4] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'13)*. ACM, New York, NY, 33–52.
- [5] Yangjun Chen, Duren Che, and Karl Aberer. 2002. On the efficient evaluation of relaxed queries in biological databases. In *Proceedings of the 11th International Conference on Information and Knowledge Management*. ACM, New York, NY, 227–236.
- [6] Cristiana Chitic and Daniela Rosu. 2004. On validation of XML streams using finite state machines. In *Proceedings of the 7th International Workshop on the Web and Databases: Colocated with ACM SIGMOD/PODS 2004*. ACM, New York, NY, 85–90.
- [7] Mihir R. Choudhury and Kartik Mohanram. 2007. Accurate and scalable reliability analysis of logic circuits. In *Proceedings of the Conference on Design, Automation, and Test in Europe*. 1454–1459.
- [8] Jeffrey J. Cook and Craig Zilles. 2008. A characterization of instruction-level error derating and its implications for error detection. In *Proceedings of the 2008 IEEE International Conference on Dependable Systems and Networks with FTCS and DCC (DSN'08)*. IEEE, Los Alamitos, CA, 482–491.
- [9] Philip Simon Dauber. 1965. An analysis of errors in finite automata. *Information and Control* 8, 3 (1965), 295–303.
- [10] Marc de Kruijff, Shouu Nomura, and Karthikeyan Sankaralingam. 2010. Relax: An architectural framework for software recovery of hardware faults. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 497–508.
- [11] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (2014), 3088–3098.
- [12] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Architecture support for disciplined approximate programming. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, 301–312.
- [13] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 449–460.
- [14] Laurent Falquet, Marco Pagni, Philipp Bucher, Nicolas Hulo, Christian J. A. Sigrist, Kay Hofmann, and Amos Bairoch. 2002. The PROSITE database, its status in 2002. *Nucleic Acids Research* 30, 1 (2002), 235–238.
- [15] Bo Fang, Qining Lu, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. 2016. ePVF: An enhanced program vulnerability factor methodology for cross-layer resilience analysis. In *Proceedings of the 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'16)*. IEEE, Los Alamitos, CA, 168–179.
- [16] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: Probabilistic soft error reliability on the cheap. *ACM SIGARCH Computer Architecture News* 38 (2010), 385–396.
- [17] Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, Gianni Antichi, and Andrea Di Pietro. 2008. An improved DFA for fast regular expression matching. *ACM SIGCOMM Computer Communication Review* 38, 5 (2008), 29–40.
- [18] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. 2003. Processing XML streams with deterministic automata. In *Proceedings of the International Conference on Database Theory*. 173–189.
- [19] Weining Gu, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Zhenyu Yang. 2003. Characterization of Linux kernel behavior under errors. In *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE, Los Alamitos, CA, 459.

- [20] Jie Han and Michael Orshansky. 2013. Approximate computing: An emerging paradigm for energy-efficient design. In *Proceedings of the 2013 18th IEEE European Test Symposium (ETS'13)*. IEEE, Los Alamitos, CA, 1–6.
- [21] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. 2012. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. *ACM SIGPLAN Notices* 47 (2012), 123–134.
- [22] Bingsheng He. 2014. When data management systems meet approximate hardware: Challenges and opportunities. *Proceedings of the VLDB Endowment* 7, 10 (2014), 877–880.
- [23] Jan Holub and Stanislav Štekr. 2009. On parallel implementations of deterministic finite automata. In *Implementation and Application of Automata*. Springer, 54–64.
- [24] Peng Jiang and Gagan Agrawal. 2017. Combining SIMD and many/multi-core parallelism for finite state machines with enumerative speculation. *ACM SIGPLAN Notices* 52, 8 (2017), 179–191.
- [25] Jarkko Kari. 2003. Synchronizing finite automata on Eulerian digraphs. *Theoretical Computer Science* 295, 1 (2003), 223–232.
- [26] Randy H. Katz and Gaetano Borriello. 2005. *Contemporary Logic Design* (2nd ed.). Pearson.
- [27] Shmuel Tomi Klein and Yair Wiseman. 2003. Parallel Huffman decoding with applications to JPEG files. *Computer Journal* 46, 5 (2003), 487–497.
- [28] Smita Krishnaswamy, George F. Viamontes, Igor L. Markov, and John P. Hayes. 2005. Accurate reliability evaluation and enhancement via probabilistic transfer matrices. In *Proceedings of the Conference on Design, Automation, and Test in Europe—Volume 1*. IEEE, Los Alamitos, CA, 282–287.
- [29] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. 2006. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *ACM SIGCOMM Computer Communication Review* 36 (2006), 339–350.
- [30] Guanpeng Li and Karthik Pattabiraman. 2018. Modeling input-dependent error propagation in programs. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'18)*. IEEE, Los Alamitos, CA.
- [31] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan, and Timothy Tsai. 2018. Modeling soft-error propagation in programs. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'18)*. IEEE, Los Alamitos, CA.
- [32] Xuanhua Li and Donald Yeung. 2007. Application-level correctness and its impact on fault tolerance. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07)*. IEEE, Los Alamitos, CA, 181–192.
- [33] Hongyuan Liu, Mohamed Ibrahim, Onur Kayiran, Sreepathi Pai, and Adwait Jog. 2018. Architectural support for efficient large-scale automata processing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*.
- [34] Tobias Marschall and Sven Rahmann. 2009. Efficient exact motif discovery. *Bioinformatics* 25, 12 (2009), i356–i364.
- [35] Jin Miao, Andreas Gerstlauer, and Michael Orshansky. 2013. Approximate logic synthesis under general error magnitude and frequency constraints. In *Proceedings of the 2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'13)*. IEEE, Los Alamitos, CA, 779–786.
- [36] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'14)*. ACM, New York, NY, 309–328.
- [37] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*. IEEE, Los Alamitos, CA, 29–40.
- [38] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. 2014. Data-parallel finite-state machines. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, New York, NY, 529–542.
- [39] Frances Perry, Lester Mackey, George A. Reis, Jay Ligatti, David I. August, and David Walker. 2007. Fault-tolerant typed assembly language. *ACM SIGPLAN Notices* 42 (2007), 42–53.
- [40] Junqiao Qiu, Zhijia Zhao, and Bin Ren. 2016. MicroSpec: Speculation-centric fine-grained parallelization for FSM computations. In *Proceedings of the 2016 International Conference on Parallel Architecture and Compilation Techniques (PACT'16)*. IEEE, Los Alamitos, CA, 221–233.
- [41] Junqiao Qiu, Zhijia Zhao, Bo Wu, Abhinav Vishnu, and Shuaiwen Leon Song. 2017. Enabling scalability-sensitive speculative parallelization for FSM computations. In *Proceedings of the International Conference on Supercomputing*. ACM, New York, NY, 2.

- [42] Ashish Ranjan, Swagath Venkataramani, Xuanyao Fong, Kaushik Roy, and Anand Raghunathan. 2015. Approximate storage for energy efficient spintronic memories. In *Proceedings of the 52nd Annual Design Automation Conference (DAC'15)*. ACM, New York, NY, Article 195, 6 pages.
- [43] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. 2005. SWIFT: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE, Los Alamitos, CA, 243–254.
- [44] Michael Ringenburt, Adrian Sampson, Isaac Ackerman, Luis Ceze, and Dan Grossman. 2015. Monitoring and debugging the quality of results in approximate programs. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, NY, 399–411.
- [45] Indranil Roy and Srinivas Aluru. 2014. Finding motifs in biological sequences using the Micron Automata Processor. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, Los Alamitos, CA, 415–424.
- [46] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, New York, NY, 164–174.
- [47] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. 2013. Approximate storage in solid-state memories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, 25–36.
- [48] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. 2002. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN'02)*. IEEE, Los Alamitos, CA, 389–398.
- [49] Wasuwee Sodsong, Jingun Hong, Seongwook Chung, Yeongkyu Lim, Shin-Dug Kim, and Bernd Burgstaller. 2016. Dynamic partitioning-based JPEG decompression on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 28, 2 (2016), 517–536.
- [50] Vilas Sridharan and David R. Kaeli. 2009. Eliminating microarchitectural dependency from architectural vulnerability. In *Proceedings of the 2009 IEEE 15th International Symposium on High Performance Computer Architecture (IIPCA'09)*. IEEE, Los Alamitos, CA, 117–128.
- [51] Arun Subramaniyan and Reetuparna Das. 2017. Parallel automata processor. In *Proceedings of the 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA'17)*. IEEE, Los Alamitos, CA, 600–612.
- [52] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. 2017. Cache automaton. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, New York, NY, 259–272.
- [53] Swagath Venkataramani, Amit Sabne, Vivek Kozhikkottu, Kaushik Roy, and Anand Raghunathan. 2012. SALSA: Systematic logic synthesis of approximate circuits. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*. ACM, New York, NY, 796–801.
- [54] Xin Victoria Wang, Natalie Blades, Jie Ding, Razvan Sultana, and Giovanni Parmigiani. 2012. Estimation of sequencing error rates in short reads. *BMC Bioinformatics* 13, 1 (2012), 185.
- [55] Li Yu, Dong Li, Sparsh Mittal, and Jeffrey S. Vetter. 2014. Quantitatively modeling application resilience with the data vulnerability factor. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE, Los Alamitos, CA, 695–706.
- [56] Zhijia Zhao and Xipeng Shen. 2015. On-the-fly principled speculation for FSM parallelization. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, NY, 619–630.
- [57] Zhijia Zhao, Bo Wu, and Xipeng Shen. 2014. Challenging the “Embarrassingly Sequential”: Parallelizing finite state machine-based computations through principled speculation. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, New York, NY, 543–558.

Received June 2019; revised November 2019; accepted December 2019