Aarohi: Making Real-Time Node Failure Prediction Feasible

Anwesha Das[1], Frank Mueller[1], Barry Rountree[2]
[1]*North Carolina State University*, adas4@ncsu.edu, mueller@cs.ncsu.edu
[2]*Lawrence Livermore National Laboratory*, rountree4@llnl.gov

***Abstract*—Large-scale production systems are well known to encounter node failures, which affect compute capacity and energy. Both in HPC systems and enterprise data centers, combating failures is becoming challenging with increasing hardware and software complexity. Several data mining solutions of logs have been investigated in the context of anomaly detection in such systems. However, with subsequent proactive failure mitigation, the existing log mining solutions are not sufficiently fast for real-time anomaly detection. Machine learning (ML)-based training can produce high accuracy but the inference scheme needs to be enhanced with rapid parsers to assess anomalies in real-time. This work tackles online anomaly prediction in computing systems by exploiting context free grammar-based rapid event analysis.**

We present our framework *Aarohi*[1], which describes an effective way to predict failures online. Aarohi is designed to be generic and scalable making it suitable as a real-time predictor. Aarohi obtains more than 3 minutes lead times to node failures with an average of 0.31 msecs prediction time for a chain length of 18. The overall improvement obtained w.r.t. the existing state-of-the-art is over a factor of 27.4×. Our compiler-based approach provides new research directions for lead time optimization with a significant prediction speedup required for the deployment of proactive fault tolerant solutions in practice.

***Keywords*-Online Prediction; HPC; Node Failures; Parsing**

I. Introduction

The research community has solved pertinent problems related to failure prediction for enhanced system reliability. Even for the contemporary HPC clusters such as Cray systems (e.g., Titan, Trinity) unstructured log mining-based failure characterization [1], [2] has been investigated. Be it system failures, such as memory or DRAM errors [3], [4], hardware failures [5], software bugs [6], GPU errors [7] or application failures [8], the past decade has contributed to automated system resilience via ML and gaining a better understanding of system logs. Such field data analysis has revealed interesting statistical properties of log events including failure distributions [9], their spatio-temporal correlations [6], the effect of temperature and power consumption on reliability and performance [10] to identify how faults manifest that lead to failures, their repair times, and root causes. Consequently, ML- and deep learning (DL)-based anomaly detection solutions have been formulated [11]–[17] incorporating techniques such as clustering, SVM (support vector machines), PCA/ICA (principal/independent component analysis), Bayesian models, decision trees, signature extraction, and neural networks.

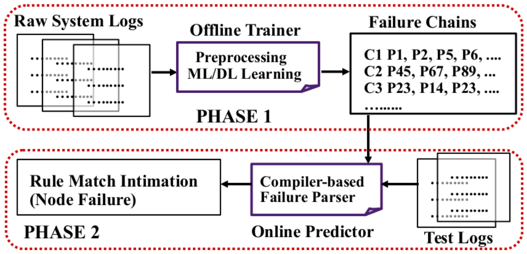

Fig. 1. Two Phase Failure Prediction

However, the efforts invested in unveiling the wealth of information from ML-based studies will pay off only when we take steps to build realistic frameworks for online failure prediction such that the overhead of costly checkpoint/restarts and wastage of compute capacity due to recalculations and waiting is reduced. In the upcoming exascale era with a quintillion (10^{18}) floating point operations per second[2], scaling the proposed solutions to work efficiently is critical. Vendors are expected to handle higher failure rates with decreasing mean time between failures (MTBF) in the order of minutes [18]. Moreover, higher component density (e.g., 10^5 nodes with GPUs and 1 TB RAM each), which require a shorter optimal checkpoint interval [19], and constantly evolving system architecture [1] impose challenges for timely resilience in practice.

To this end, we propose Aarohi, an efficient *node failure predictor* that prevents impeding failures of computing systems online and in real-time. Aarohi predicts failures by analyzing logs on average in 0.31 msecs for a phrase chain of length 18, i.e., a speedup of over 27.4× w.r.t. the existing state-of-the-art [16], [20]. Our solution is applicable to any ML-based pre-trained chain of events leading to a node failure. Chain construction with precursory events of propagating anomalies for failure detection has been demonstrated for HPC systems [21]. Even with an improved precision [14]–[16] obtained by ML-based solutions, the inference time (not mentioned in some works [14], [22]) may not be short enough to enable real-time prediction. This paper focuses on transitioning from an offline trainer irrespective of its algorithmic complexity to a scalable, adaptive online predictor. This predictor is automatically generated from the specifications obtained during training and, due to this automation, even may be dynamically updated if new training data becomes available. Figure 1 shows the two phases of failure prediction. First, the offline training phase of logs achieves high recall of failure chains. Second, the online prediction phase strives to

[1] *Aarohi* means *ascending* in the *Sanskrit* language. It personifies the gradual event-wise progression towards successful failure prediction.

[2] ECP: https://www.exascaleproject.org/what-is-exascale/

enhance the inference speedup while achieving sufficient lead times to failure. This paper is not about Phase 1, Aarohi's novelty is in automatically generating an inference tool for Phase 2 based on failure patterns identified in Phase 1.

II. Background

Once failure indicators have been trained, inferring impending failures from the new test data is not fast enough to aid real-time prediction. Even though improved learning-based solutions exist, practical deployment of such schemes for successful online prediction requires fast mechanisms to reduce the failure inference time for proactive fault tolerance. As an example, DeepLog [16] and Cloudseer [20] incur 1.06 and 2.36 msecs, respectively, to check a single log message for online detection using techniques of LSTM and automatons. For a failure chain of length 10, they might require as much as ≈10.6 and ≈23.6 msecs, respectively. Yet log messages can be as low as μsecs apart in time. Can we predict anomalies any faster? To what extent? We address this challenge by automatically generating an online predictor from training-derived event patterns in an adaptive, generic and fast manner.

Challenges: While many failure prediction solutions have been proposed, most of them cannot be used online [12] to take timely proactive recovery actions (e.g., job migration [23], process cloning [24]). Some of the hurdles for real-time failure prediction in large-scale computing systems are:
1. ML-based schemes are effective offline trainers, but their analysis speed is unsuitable for real-time failure mitigation. Even with accurate learning and acceptable lead times to failures, slow inference with insufficient speed may fail to finish proactive actions before the fault freezes a component.
2. The pace of analyzing incoming event logs by the predictor should be compatible to the inter-arrival times of the consecutive system logs (e.g., msecs or μsecs).
3. An online predictor should be reusable with evolving event patterns and diverse system types. It should accommodate software and logging paradigm variations with minimal overhead, without receding efficacy over time. This is non trivial since upgrades and new systems introduce new features [1] and unseen log messages, thus new failure patterns emerge. Apart from re-training, a robust predictor needs to be adaptive and portable so that the core prediction scheme remains functional and the approach becomes sustainable across systems.

Table I illustrates that there exist differences in logs obtained from Cray XC versus XE systems. In fact, even within the Cray XC series (XC40/XC30), different HPC sites can produce different logs based on their specific hardware and software. This elucidates that log upgrades are common, and even similar systems belonging to the same family (IBM, Cray) possess variations in their logs since they contain vendor-specific templates [1] and feature diverse rates of logging.

Contributions: Aarohi automatically generates a fully unsupervised parser from a DL-based training. It provides significantly faster failure prediction via novel parsing of phrase sequences. This paper makes the following contributions:
1. We propose an efficient predictor, which can proactively flag failures in online streamed test data using grammar-based rules. The predictor works with trained failure chains, which are confirmed patterns of node failures (derived in consultation with experts and system administrators).

TABLE I
Log Variations

Features	**Cray XC40**	**Cray XE**	**Cray XC30**
Processor	Haswell, KNL	AMD Opteron	Haswell, IvyBridge
Burst Buffer, Job Scheduler	Yes, Slurm	No, Torque	No, Slurm
Interconnect	Aries (DragonFly)	Gemini (Torus)	Aries (DragonFly)
System Log Data	Controller (bcsysd), Boot-logs, SEDC differ from XE	Controller (syslog-ng), Boot-logs, SEDC differ from XC	Controller (bcsysd), Boot-logs, SEDC differ from XE

2. We describe the process for translating a set of failure chains identified via ML for a system to a rule set. This is a generic approach that can be adapted to any system (with specific failure definitions) automating the process of rule generation.
3. We illustrate how our predictor adapts and incurs low overhead for log variations across diverse system types demonstrating robustness for cross-system portability.

TABLE II
System Logs

System	**Time Span**	**Size**	**Scale**	**Type**
HPC1	5 months	150GB	5576 nodes	Cray XC30
HPC2	6 months	98GB	6400 nodes	Cray XE6
HPC3	8 months	27GB	1630 nodes	Cray XC40
HPC4	6 months	15GB	1872 nodes	Cray XC40/30

Log Details: The system logs used for the study have been obtained from 4 HPC systems. Table II enumerates the system details including the duration of log collection, size and system scale. These are production HPC clusters serving millions of compute node hours. Offline training uses these logs to identify node failure patterns for later online prediction. Once patterns of failure chains are learned, we discuss how Aarohi infers failures from the test data efficiently for proactive counter measures to be completed before a node seizes to respond.

Node Failures: Aarohi predicts node failures. Node failures are anomalous node outages caused by hardware, software or application malfunctioning. We exclude intentional node shutdowns that are maintenance related or periodic shutdowns triggered by the operator. We further confirmed normal symptoms and abnormal ones, i.e., *failed messages* in the logs, by consulting with the system administrators. Root cause diagnosis and the intricacies of DL-based training are not the main focus of this paper. We build on prior work that identifies failure chains [20], [25]. The novelty lies in the second phase, i.e., the automatic generation of the prediction engine.

III. Online Prediction Design

Figure 2 depicts the overall design of the failure prediction scheme, Phase 1 for offline learning followed by Phase 2 for online prediction. We briefly outline the offline training phase (obtained from [25], see there for details) used before detailing the online prediction scheme, which is our contribution.

DL-based Training (Phase 1): The following steps summarize an LSTM-based approach to learn node failure chains:
1. The logs are trained using LSTM to learn the message

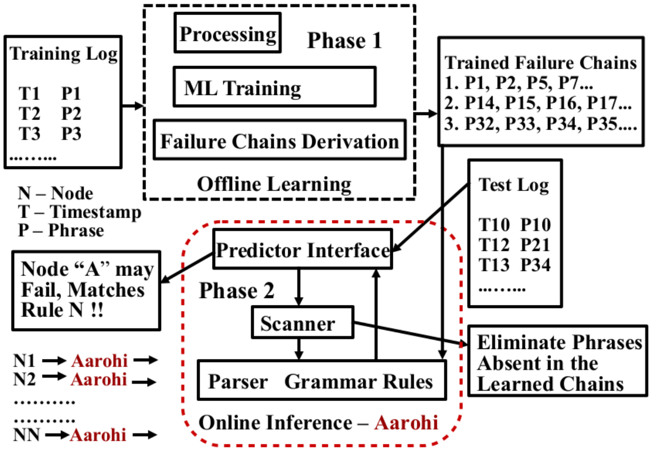

Fig. 2. Overall Design

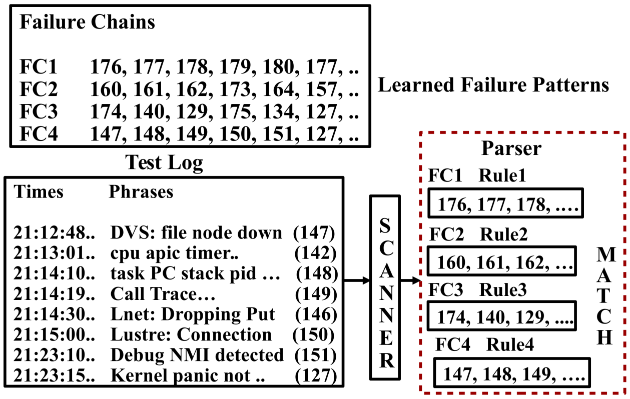
Failure Chains
FC1 176, 177, 178, 179, 180, 177, ..
FC2 160, 161, 162, 173, 164, 157, ..
FC3 174, 140, 129, 175, 134, 127, ..
FC4 147, 148, 149, 150, 151, 127, ..
Learned Failure Patterns
Test Log
Times Phrases
21:12:48.. DVS: file node down (147)
21:13:01.. cpu apic timer.. (142)
21:14:10.. task PC stack pid … (148)
21:14:19.. Call Trace… (149)
21:14:30.. Lnet: Dropping Put (146)
21:15:00.. Lustre: Connection (150)
21:23:10.. Debug NMI detected (151)
21:23:15.. Kernel panic not .. (127)
SCANNER
Parser
FC1 Rule1
176, 177, 178, ….
FC2 Rule2
160, 161, 162, …
FC3 Rule3
174, 140, 129, ….
FC4 Rule4
147, 148, 149, ….
MATCH

Fig. 3. Aarohi Design

patterns based on the history of the training data.

2. The messages that are definitely not benign (e.g., erroneous or unknown) along with *failed messages* (e.g., cb_node_unavailable) corresponding to anomalous node shutdowns are segregated a priori. Based on this phrase labeling, sequences of events that lead to node failures are formed from Step 1. The rest of the phrases are omitted from consideration while forming the failure chains. Thus, we have node failure chains (FCs) composed of anomaly relevant phrases.

Table III shows 6 phrases leading to a node failure (FC3 of Fig. 3) of which P1, P5, and P6 are erroneous while the rest are unknown. All messages are pertaining to a specific node (e.g., c0-0c2s0n2), but the node identifiers are removed here for brevity. For a specific node, Table III depicts the calculated ΔTs in the 3rd column computed from the adjacent phrases. We use the discussed LSTM-based training methodology [25] for Phase 1. Notice that any learning technique will work as long as the predictor can be fed with a sequence of coherent phrases leading to failures, i.e., our approach is not dependent on LSTM, rather it works generally for failure chains. For a specific system, how a failure is defined, a chain is formed, and what technique is used for deriving the chain may vary and does not affect the predictor. Of course, higher accuracy of the failure chains implies better prediction efficacy.

Predictor Design (Phase 2): The online prediction method, Aarohi, consists of the following steps:

1. The incoming stream of log events with their timestamps are scanned through regular expressions (RE) via the auto-generated rules of the scanner. These phrases are tokenized and the events that do not appear in any of the trained FCs are skipped as those are of no interest to the predictor.

2. The parser consists of rules expressed in a context free grammar (CFG), which are directly and automatically generated from the learned FCs (Phase 1). Incoming events are checked against these rules to predict future failures. The rules are formulated based on the message sequence and the time difference between two adjacent events. The latter captures the contextual relevance of events over time. This token-based rule check is fast and is the source of our inference speedup.

For each node in the cluster, we dedicate a predictor instance that processes messages of that node only (see Fig. 2). The Token column in Table III illustrates the handling of tokens corresponding to the timestamps and phrases. Figure 3 demonstrates how multiple rules of parsers are checked for diverse FCs over a stream of log events. The CFG rules are automatically derived from the learned FCs (e.g., FC1 - FC4). The test data is tokenized, and phrases not appearing in the set of learned FCs are removed (i.e., 142 & 146). Then, a specific rule is selected based on the starting phrase (token match). The sequence matches FC3 in Tab. III. This continues until one of the two conditions are met: a) no more phrases are left in the test data, or b) a match has been found. If a match is found before the test data ends, the parser resets, beginning with the first event phrase seen after the last phrase of the matched FC.

From Failure Chains to Rules: Table IV defines a CFG G with a set of non-terminal symbols (N), terminal symbols (T), start symbol (S), and production rules (P). The start symbol is a non-terminal and leads to productions with leading terminals of FCs. The context free productions (P) are a subset of rules (R) that are formed as a union of non-terminal and terminal symbols. We have 1 lookahead, i.e., every phrase in the sequence is checked one at a time to decide on the parser action and select a production. Figure 4 highlights certain features of the observed sequence of phrases (FCs):

TABLE IV
PARSER GRAMMAR

Notations	Meanings
G = (N, T, P, S)	LALR(1), 1 Lookahead, Start Symbol S
N	Non-Terminal Symbols
T	Terminal Symbols
P	$R=N \cup T$, $P \subseteq R^{+}$ (Production Rules)
FC1 FC5	(176 177 178 179 180 137) (172 177 178 193 137)
P_FC	S: (176 177 178 179 180 137) | (172 177 178 193 137)
P_LALR	S→(176 C 137) | (172 C 137) C→(B 179 180) | (B 193), B→(177 178)

1. FCs usually have short subchain matches (e.g., 177 & 178 in FC1 and FC5), and they may end with a common failed message (e.g., 7 in C and D).

2. The ΔTs (time difference) between adjacent phrases are usually $<$ 2 mins. Figure 5 depicts the cumulative phrase arrivals for nodes A and B on a log scale with inter-arrival times in msecs. For A, 92.05% (278) phrase arrivals have $\leq$ 2

TABLE III
LOG MESSAGE PROCESSING

Timestamp	Phrase	ΔT (secs)	Token
04:58:57.640 (T1)	[Firmware Bug]: powernow k8: * (P1) E	0	<T1 174>
04:59:06.317 (T2)	DVS: verify filesystem: * (P2) U	8.323 (T2-T1)	<T2 140>
05:00:26.823 (T3)	DVS: file node down: * (P3) U	16.506 (T3-T2)	<T3 129>
05:00:51.669 (T4)	Lustre: * cannot find peer * (P4) U	24.846 (T4-T3)	<T4 175>
05:01:14.297 (T5)	Lnet: critical hardware error: * (P5) E	36.372 (T5-T4)	<T5 134>
05:03:24.403 (T6)	cb node unavailable: (P6) E	130.106 (T6-T5)	<T6 127>

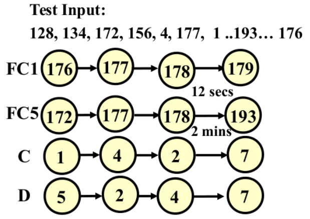

Fig. 4. Chains

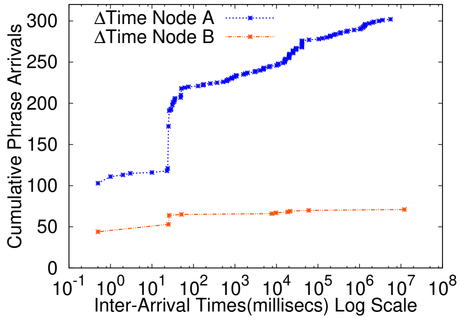

Fig. 5. ΔTimes

mins ΔTs (order of μ & msecs) in sample data of 302 phrases, and ≈13 of them have ΔTs $\geq$ 17 mins. For B, 98.6% arrivals happen within $\leq$ 1.1 mins in sample data of 71 phrases. For a single day, while B's logs spanned across ≈3.5 hours, A's logs stretched ≈8.75 hours (hence, the higher count). Similar trends are observed at other time frames for other nodes as well. As seen, sometimes there exist steep rises in cumulative arrivals for certain ΔTs (e.g., A: 51 & 19, B: 9 & 11 arrivals for ΔTs of 25 & 26 msecs, respectively). The routing latency from a remote service can cause intermittent delays in phrase arrivals within a burst of messages from the same log source. The filesystem or interconnect related delays can be caused by various environmental factors. A defined timeout can help identify unexpected delays during parsing (e.g., 4 mins when 93% of the phrase inter-arrival times are $\leq$ 4 mins) based on such observed ΔTs (checked by semantic actions).

3. Common subchains exist but the starting phrase is usually different for FCs. Few common phrases or swaps (e.g., 2 & 4 in C and D) may occur in sequences of phrases over time.

The chain of terminal symbols leading to an accept state is the distinguishing feature of any system-defined failures. To clarify, we used *failed message* earlier to refer to typical node shutdown messages (e.g., P6 in Table III). The chain of terminal symbols of our grammar, G, refers to any relevant message of an FC (e.g., 172, 177, ...in FC5), not just the last phrase of an FC (e.g. 177, 157, 127 of FC1 - FC4 in Fig. 3). From these chains (e.g., FC1 and FC5 in Fig. 4), the non-terminals are derived automatically by combining common sequences of terminals (e.g., B→(177 178) of P_LALR in Table IV). Since prefixes and common phrases exist in failure chains, we formalize our parser as an LALR(1) [26] grammar. The derivation of grammar rules is shown in Table IV (P_FC and P_LALR) for chains FC1 and FC5.

Tokenization: During failure inference, logs are processed a single line (event) at a time and parsed adhering to the FC templates from Phase 1. As an example, consider the following two phrases from the test logs:

P1: *DVS: verify filesystem: file system magic value 0x6969 retrieved from server c4-2c0s0n2 for /global/scratch does not match expected value 0x47504653: excluding server*
P2: *pcieport 0000:00:03.0: [12] Replay Timer Timeout*

The first input phrase is parsed until it reaches *DVS: verify filesystem:*, which matches a template of a phrase pertaining to some FC (e.g., P2 in Table III). The remaining content has variable components, such as `0x6969`, node id `c4-2c0s0n2`, or directory path `/global/scratch`, none of which are further considered. Based on this template match, the corresponding token (say 140) is used for matching parser rules. For the second phrase, *pcieport... Timeout*, the parser checks up to the end of lexical rules, finds no match with any FC-related template, and discards it. Raw log tokenization and rule check-based inference are closely integrated in Aarohi, unlike prior online log parsers such as Spell or Drain [27]. Similarly, the remaining templates of the FCs are considered by parsing the tokens emitted by the lexer when receiving log input and then used to flag predicted node failures.

Consider Figure 4 for a test input 128, 134...4,..., 176. The scanner discards phrases 128, 134 and similar unrelated messages during tokenization since they do not match any of the FC-related phrase templates. Next, phrase 172 matches the starting phrase of FC5. Hence, that rule, 172, 177, 178..., is checked. The parser continues until a mismatch is encountered when it finds 4, another event that is tokenized as it occurs in chains C and D, yet it expected 177 as the next token (phrase) as per FC5. The parser skips such mismatches and continues parsing until a ΔT-based threshold violation occurs, or it reaches the end of test log. This is important as the test data can contain messages not part of any FC. We transform prefixes into a singular non-terminal production rule up to the end of the common substring. Based on left-to-right token matches, a matching rule is parsed until a rule is fully matched, i.e., at token 193 in the example. A regular parser would exit after reaching this accept state. Our parsing harness, however, proceeds with the next token, 176, and invokes another instant, in this case to check if rule FC1 matches.

Interleaved Rule Matches: In theory, an incoming set of phrases can match any of the failure patterns recognized in the past. This necessitates the need to simultaneously check for multiple FCs because log events of a single node can match a rule *partially* when tokens pertaining to another rule may be encountered. Aarohi's set of rules each match a unique FC with the ability to start/stop an FC based on which event (token) is encountered next. While evaluating a specific rule, say FC5, during testing, the following cases may occur:

1. The first phrase of another rule (say 1 of C) matches the incoming phrase, but the parser continues to check FC5. If the test data does not match FC5 fully (only partially) but could have matched C, then Aarohi misses this C-match, which could result in a false negative. This is the case for any partial rule match token-wise interleaved with another full rule match.
2. Tokens may be intertwined across rules, say by alternating tokens from FC5 and C, such that both rules could be matched. Here, only the first rule with a token match results in an accept while the other rule is never parsed, also resulting in a false negative. However, the first match already indicates a failure and thus subsumes a subsequent failure during the same time frame, i.e., the false positive of C is irrelevant for our application scenario for these latent node failures. Notice that there are no cases where this method results in false positives.

TABLE V
MULTIPLE RULE MATCHES

System	Duration	Missed Rules	Interleaved	#Nodes
HPC1	4 mons	No	Yes	23
HPC2	3 mons	No	Yes	19
HPC3	3 mons	No	Yes	15
HPC4	4 mons	No	Yes	20

In practice, we found that case 1 does not occur in the inspected test logs (see Table V) but case 2 is seen, e.g., interleaved tokens from FC5 & C are observed. Although, case 1 is theoretically possible, it was not observed because:
1. Healthy node events tend to present a mismatch for FCs pertaining to failures. Unhealthy nodes experience a complete match with FCs with only rare cases of interleaving. The earlier a rule matches, the larger will be the remaining time for proactive measures irrespective of future rule matches.
2. Our inspection shows that the first rule tends to match fully, and later phrases do not tend to lead to cases with true positives (i.e., complete FC match). Occasional interleavings exist (Table V), but once a specific rule has been partially matched, it tends to be safe to skip subsequent rules.

Table V provides empirical evidence of the absence of cases where multiple rules match completely in close temporal proximity (with interleaving) in our data. In the systems studied, either unhealthy node logs do match the FCs or, once an FC match starts, other interleaved FCs do not result in a false negative. However, inherent in any training-based scheme, dynamic re-training and regeneration of FCs is necessary if a new failure pattern evolves over time in the test data. More commonly, different nodes may fail simultaneously in time when matching the same FCs, or a node may fail successively over *different time frames*. This substantiates that our scheme suffices, and that we are not missing imminent failures. Since partial matches do not enhance Aarohi's resilience capability, we chose a simple, yet effective inference model implemented by this parsing methodology.

Algorithm 1 enumerates the steps in automatically translating a generic set of FCs to parser rules. This translation is performed offline, i.e., its time complexity is not the critical path. Assume we have a set of FCs from the output of Phase 1 training for an HPC cluster. The distinct phrase templates (e.g., Phrase column in Tab. III) encompassing all the FCs are enumerated uniquely. Each phrase ID is then tokenized by assigning (#5) a global token (e.g., $\{101\ 102 \ldots\} \rightarrow \{\text{P1 P2} \ldots\}$) forming a token list. Unique rules are formed with the corresponding tokens (#6) based on the sequence of phrases in the FCs, such as:
FC1: $\{123\ 135 \ldots\} \rightarrow$ R1: $\{\text{P23 P35} \ldots\}$,
FC2: $\{141\ 152 \ldots\} \rightarrow$ R1: $\{\text{P41 P52} \ldots\} \ldots$

With a similar single chain rule set (#8), recursive rules (#15, #16) can be derived by substituting subchains (#14), if any, between multiple rules. These subchains form the non-terminals of the LALR(1) grammar (see P_LALR in Table IV).

Algorithm 1: From FCs To Parser Rules

```
input : FC List
output: Rule List
1  $T \leftarrow \emptyset, S \leftarrow \emptyset$ // $T \leftarrow$ Token List, $S \leftarrow$ Rule List
2  foreach ($FC \in$ FC List) do // Failure Chain
3      $R \leftarrow \emptyset$
4      foreach ($Phrase \in FC$) do // Form Token List
5          if ($Phrase \notin T$) then $T \leftarrow T \cup Phrase$
6          $R \leftarrow R \cup Phrase$ // Unique Chain Rule
7      end
8      $S \leftarrow S \cup R$ // Rule List from Unique Rules
9  end
10 // Derive LALR(1) Rules from Rule List S
11 foreach ($V \in S$) do
12     foreach ($U \in S$) do
13         if ($V \neq U$) then // Substitute Subchain
14             foreach ($C =$ Subchain$(U, V)$) do
15                 $V' \leftarrow head(V) \cup C \cup tail(V)$
16                 $U' \leftarrow head(U) \cup C \cup tail(U)$
17                 $S \leftarrow (S \setminus \{U, V\} \cup \{U', V'\})$ // Update S
18             end
19         end
20     end
21 end
```

Algorithm 2: Aarohi Prediction

```
input : Test Data, Token List T & Rule List S from Algo. 1
output: Matched FC Rule
1  // Online Inference
2  while (Test Data$\neq$ NULL) do // Incoming Phrase
3      Parse(Test Data) // Call the parser
4  end
5  // Parser Rules
6  $Token \leftarrow Phrase$ // Tokenize
7  if (Token $\in$ T) then return Token + Arrival Time
     // Relevant Token
8  else Skip Token (not relevant)
9  $R^{+} \leftarrow$ S // Rule List, e.g., R1
10 P1 error P2 error P3... $\leftarrow$ Sequence Matched Rule 9
11 if ($\exists$ error) then // Mismatch while parsing
12     if ($\Delta$T$\leq$Timeout) then Skip Token, Continue
13     else Reset after Current Token $\#P$ // Restart
14 end
15 if (Test Data$\neq$ NULL) then // On a Reset
16     Parse(Test Data) // Start after Token $\#P$
17 end
```

Algorithm 2 encapsulates Aarohi's operation during test data inference. Once the parser is invoked, each phrase in the

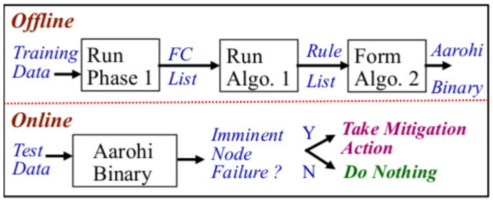

Fig. 6. Offline Training to Online Testing

test input is tokenized (#6) to check if it matches any of the tokens of the Token List generated from Algorithm 1. On a match, the token with its arrival time is sent to the parser (#7), else it is irrelevant and discarded (#8). For parsing, the Rule List obtained from Algorithm 1 is used with suitable semantic actions (R^+, #9, #10) such as:

R1: {P23 P35 P45 ...}, R2: {P41 P57 P62 ...} ...

For the first match, an incoming token matches with one of the rule's first token (say rule `R1`), and that ($\texttt{R1}^{th}$) rule is checked. On an error (i.e., incoming token differs from expected token), if the difference between the current time and the last matched token's arrival time (ΔT) exceeds a predefined threshold, the parser aborts (#13), else it continues (#12), because, in practice, inordinate delays between incoming phrases of known failure chains do not belong to the same failure pattern. Similar semantic actions continue until a reset is triggered or no more phrases are left in the test log. Skipping tokens (#12) is essential for rule checking to discard the non-relevant phrases in between FC-related phrases. Multiple rule matches may occur back-to-back in an input stream. For any remaining unprocessed phrases in the test data, the online prediction continues from the phrase appearing after the last phrase of rule R1 in the Test data (after a match, #10) or the last processed token before reset (#13). This heuristic's complexity is linear in input size (i.e., log file) and, together with grammar-based parsing, aids in Aarohi's inference speedup w.r.t. the existing detection schemes.

Figure 6 summarizes the workflow to facilitate transitioning from offline training to online prediction for any system. Phase 1 produces FCs, which, when run with Algo. 1, produce parser rules. Algo. 2 with equivalent grammar rules, appropriate error handling, and semantic actions (R^+) produces the binary. Aarohi is then run with new test data for online prediction.

IV. Evaluation

Aarohi is implemented using the flex/bison parser in C++. Our FCs contain sparse subchain matches for which non-recursive chain rules suffice. Aarohi's token handling ensures continuation of parsing by skipping unexpected phrases in the test stream with appropriate semantics. Parsing performance is reported for an Intel quad core processor running at 2.83 GHz. For all the systems, the test log data used for prediction is different from the training data used for learning the FCs.

We report prediction times, i.e., the time taken to check if a variable length sequence of phrases (not a single log message) matches any of the FCs. The inference times are obtained with compiler optimization level O3 enabled and trace output for debugging disabled. From the timestamped node *failed message* in the test data to the event phrase at which the predictor flags match, we compute the expected lead times to imminent node failures. Aarohi's evaluation metric is speedup over accuracy in the context of real-time failure prediction.

TABLE VII
Efficiency Formulae

Formula	Implication
Recall(%)=TP/(TP+FN)	Fraction of node failures *correctly* identified
Precision(%)=TP/(TP+FP)	Fraction of node failures predicted
Accuracy(%)= (TP+TN)/(TP+FP+FN+TN)	Fraction of correct predictions in the entire set
False Negative Rate FNR(%)=FN/(TP+FN)	Rate of missed failures
True Positive (TP)	Correctly predicted failures
True Negative (TN)	Correctly rejected as not failures
False Positive (FP)	Incorrectly predicted failures
False Negative (FN)	Incorrectly rejected as not failures

Table VII lists the standard efficiency metrics with their formulas. The terms are defined in the context of node failures. Figure 7 shows recall, precision and accuracy obtained in Phase 1. For the considered node failures in each of the 4 systems, the false negative rate ranges from 5 to 17.6%. This does not affect the prediction times in Phase 2, but it is indicative of the efficacy of FC-based rules used for inference. The precision exceeds 86% in all cases.

Observation 1: *Recall, precision, and accuracy exceed 86%, 88%, and 80%, respectively, across all 4 systems with a moderate false negative rate below 18%.*

Prediction Time: Figure 8 shows the prediction times of 9 phrase chains. The test data contains phrases that exist in some FC. In this case, the parser skips a token on a mismatch unless any of the termination conditions are met. Aarohi takes 0.18 msecs to 0.6 msecs for chain lengths ranging from 5 to 50 with a std. deviation $\leq \pm 0.068$ msecs. Figure 9 depicts the prediction times with log messages that include benign phrases that are not part of any FCs (as they appear in the test logs). These get discarded by the scanner without tokenization. In such a case, Aarohi obtains inference times ranging from 0.17 to 0.56 msecs with a std. deviation $\leq \pm 0.065$ msecs. This is a realistic case containing many benign phrases and some FC-related phrases. These times are comparatively lower than the former because, in the previous case, all phrases are tokenized but later skipped by the parser during rule checking.

To understand the variation of inference times over different CPU architectures, for increasing chain lengths, we ran experiments on Intel Quad Core, Xeon, Xeon Silver and AMD Opteron platforms. Figure 10 shows that Opteron takes more time than the Intel platforms, however, with increased number of phrases the difference in prediction times between Xeon, Xeon Silver and Opteron is less than 2 msecs. Overall the std. deviation do not exceed ± 0.67 msecs. Please note that an increase in chain length or number of log messages in a sequence do not necessarily indicate higher prediction times. The individual phrase size varies (e.g., P1 & P2 under Tokenization) along with the proportion of FC-related phrases. This is why Aarohi checks a 302-length chain in less time

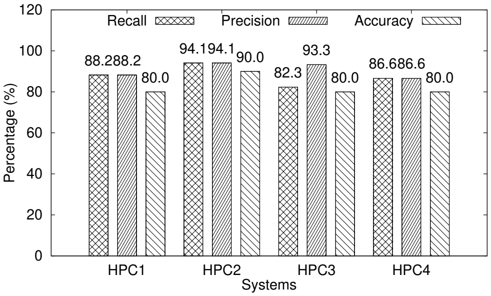

Fig. 7. Phase 1 Efficiency

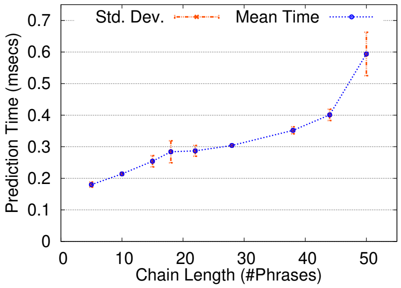

Fig. 8. Prediction Time

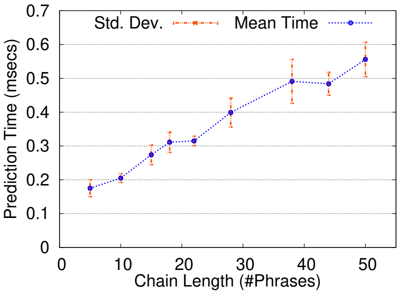

Fig. 9. With Benign Phrases

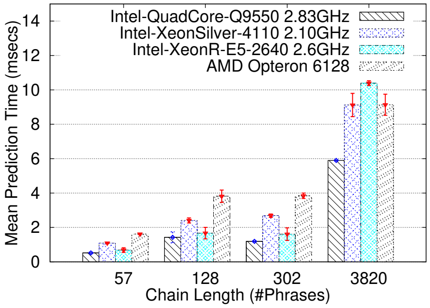

Fig. 10. Diverse Platforms

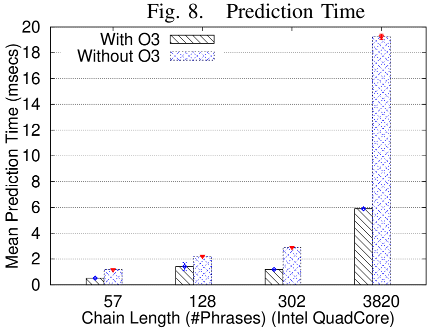

Fig. 11. Optimization

TABLE VI
SPEEDUP

Approach	**Prediction Times (msecs)**				
	Chain Lengths				
	1	10	50	128	302
Aarohi	0.05	0.205	0.556	1.427	1.1904
Desh	0.12	1.856	8.761	19.356	32.681
DeepLog	1.06	10.6	53	135.68	320.12
CloudSeer	1.81	18.1	90.5	231.68	546.62

than one of 128-length (see Tab. VI). The former contained long phrases in most lines unlike the latter, which had short phrases including call traces. Figure 11 shows the difference in prediction times with & without O3 optimization. For a 302-length chain, 58.96% improvement is observed (2.9 msecs to 1.19 msecs). For a stream of 7443 messages, with & without O3 flag takes 45 msecs and 77 msecs, respectively. Their corresponding message sizes vary between 4K & 712K.

Observation 2: *Aarohi obtains less than 11 msecs inference times across diverse CPU platforms. Such a compiler optimized approach enables rapid inference for diverse chain lengths with varying message sizes. Speedup is not linear w.r.t. the chain length or message size, hence, per log entry times are not appropriate indicators of time complexity.*

Recent anomaly detection solutions [16], [20], [25] report the testing times of a single log entry. Cloudseer [20], Desh [25] and DeepLog [16] are suitable candidates for comparison since they employ contemporary techniques, such as automatons and LSTMs for log sequence analysis. Other ML-techniques are expected to consume more time. Aarohi's metric is a chain of messages, as opposed to a single anomalous message. Table VI highlights that Aarohi is considerably faster than Desh, DeepLog and Cloudseer for increasing chain lengths. Lack of similar system logs and source code make an exact comparison difficult. However, by actually implementing Desh (as in the paper), we obtained 2×-27.4× improvements. Also, the reported times in DeepLog and CloudSeer are for single log entry checks pertaining to anomalous messages. Moreover, it is not clear if raw log tokenization time has been accounted in prior work. Differences in speedup become higher and discernible with increasing chain lengths. For a 302-length chain Aarohi is 27.4× faster than Desh (32.68 msecs to 1.19 msecs). Python-based frameworks with ML-libraries are slow runtime interpreters compared to C++, facilitating Aarohi's speedup.

Observation 3: *For chain lengths 1 to 302, Aarohi is over 27.4× faster than the current state-of-the-art approaches underlining Aarohi's potential for real-time failure prediction. The speedups increase with increasing chain lengths.*

Figure 12 shows the fraction of phrases in the test data that correspond to some FC across all the systems. As seen, most phrases are dissimilar to FC-related phrase templates. As healthy node logs dominate, their phrases are not part of any FCs. The percentages of phrases pertaining to any FC-related token range from 29.81% to 46.72%. This determines the portion of messages discarded during lexical scanning.

Observation 4: *The fraction of FC-related phrases eventually tokenized are below 47% in the test data indicating that only a minor fraction of log events need to be tokenized and then parsed during online log analysis.*

Lead Time Sensitivity: Figure 13 depicts the lead times obtained for 10 FCs before the terminal message appeared in the test data. The last phrase matched in the FC corresponds to a timestamp, which is subtracted from that of the eventual node failure message to compute the lead times. Similar lead times are obtained for other node failures. These node failures correspond to chains of length 5 to 50. With prediction times below 0.65 msecs, we can obtain effective lead times (with prediction times deducted) higher than 3 minutes. This means that for failure F5 in Fig. 13, suitable mitigation action can be taken in 3.24 mins (3.245-0.00001). The average lead time is more than 2 mins for node failures across all the 4 systems. The chain lengths do not affect the lead time. However, the ΔTs between the log messages affect the lead time. Shorter

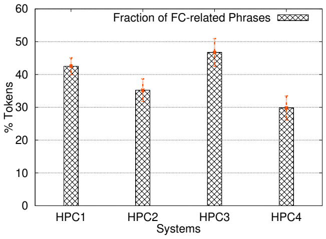

Fig. 12. Phrase Fraction

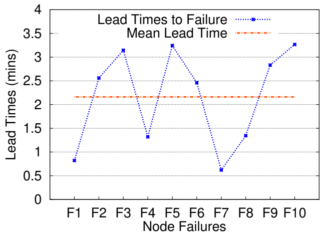

Fig. 13. Lead Times

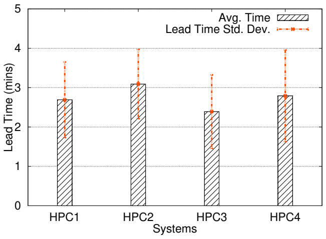

Fig. 14. System Times

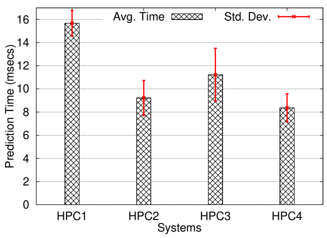

Fig. 15. Prediction Times

ΔTs in the sequence matched in a FC lead to shorter lead time since the upcoming terminal message is immanent. That does not imply that longer ΔTs indicate higher lead times. ΔTs are a refinement to prevent false positives during inference rather than to increase the lead times to failures.

Observation 5: *With Aarohi, effective lead times to failures are >3 mins with an average of 2 mins. Prediction times are <11 msecs for 3820-length. Early prediction is required to leave sufficient time for failure mitigation approaches (e.g., process migration) before a node stops responding.*

Cross-System Comparison: Figures 14 and 15 depict the avg. lead times and prediction times of node failures across all the systems. While the lead times exceed 2 mins, the avg. prediction times are less than 16 msecs for over 100-length chains. The std. deviation of prediction times is $\leq$ $\pm$2.3 msecs, which is higher than in Fig. 9. This variation is caused by the diversity of node-specific test sequences on different systems w.r.t. their corresponding FCs. The avg. lead time is $\approx$2.74 mins with a moderate std. deviation of $\leq$ $\pm$1.16 mins.

Observation 6: *Aarohi obtains > 2.3 mins avg. lead times to node failures with an avg. prediction time of no more than 16 msecs across systems. The std. deviation of prediction times are higher across systems than over different failure patterns of similar chain lengths. This is due to significant variations in the test sequences w.r.t. the FCs across systems.*

Comparative Analysis: Apart from DeepLog [16], Desh and Cloudseer [20], whose testing times are compared with Aarohi, several other researchers have studied failure prediction. Table VIII illustrates that most solutions do not perform lead time analysis [13], [14] w.r.t. the prediction time. Approaches such as GA [28], SVM, clustering etc. are not effective for online prediction in contemporary systems since they are intractable with scale. Klinkenberg et al. [22] obtain higher lead times for known node soft lockups through supervised classification, unlike generic inference of Aarohi. Although past work stresses on building online solutions [15], [28], lack of expeditious prediction mechanism pose deficiencies in practice. Aarohi's contribution can be effective for both cloud and HPC systems for proactive fault management.

Adaptability: In the production HPC clusters, the following cases are prevalent (see Table I discussion):

1. Systems of different generations have syntactically different but semantically equivalent logs (e.g., Cray XE vs. XC40).
2. Systems of the same generation with dissimilar h/w or s/w have syntactic log variations (e.g., Cray XC30 vs. XC40).
3. Software is updated on a given system after a period of time, which changes the log's syntax (e.g., several Cray systems upgraded to Slurm from Torque as their job scheduler or incorporated burst buffers, an intermediate storage layer).

In face of such software upgrades or log variations, phrase re-mappings and rule updates can suffice, without changing the overall workflow of Aarohi. Tab. IX lists 6 phrases from 4 diverse clusters of which 2 are HPC, namely a Cray XK* and a BlueGene/P, and 2 are distributed (DS) systems, namely Cassandra and Hadoop. The DS logs correspond to 2 application bugs[3]. In practice, DS logs do not have node identifiers in every log message as the logs are application-centric. Upon prior preprocessing and correlation specific to any system with its failures, we discuss Aarohi's generic adaptability.

Phase 1 training is necessary for every system. This step is a prerequisite before our predictor adapts to new FCs. As seen, certain BG/P phrases have similar meanings as Cray logs (e.g., P6 in BG/P). In such cases, phrase mappings can be updated in the scanner (e.g., XC: 7→cb node unavailable, changes to BG/P: 7→node system halted) without any change in grammar rules. While some phrases remain the same (e.g., P4 & P5 in XK & XC), few undergo minor changes (e.g., heartbeat failures for Cray XK & XC). The scanner thus requires minor updates. However, for Cassandra[4] and Hadoop[5], the FCs change due to major log variations. Since it is not enough to update the mappings as the context differs, the scanner produces new tokens and the rules have to be reformulated with the new phrase identifiers (e.g., P1 to P6 in Tab. IX).

Discussion: Let us discuss a number of important considerations for real-time failure prediction.

(1) Predictor Placement: One pertinent question arising in a large-scale cluster is: Where can an online predictor be located for efficient failure handling? Figure 16 depicts the high-level overview of HPC vs. data center facilities. Cray systems utilize an HSS manager (hardware supervisory system) to administer the chassis/blade controller that manage the compute/service nodes. Nodes link to the System Management Workstation (SMW) via managers to collect system logs from the cabinets. Aarohi can be placed on the HSS network, an aggregate workspace over the entire interconnect where logs are accessible. This helps in two ways. First, daemons running on compute nodes can affect the jobs running on the cluster.

[3] Cassandra & Hadoop logs were generated in the lab after bug reproduction, HPC 5 & 6 logs were obtained from researchers who used them in the past.

[4] Cassandra: https://issues.apache.org/jira/browse/CASSANDRA-11050?attachmentSortBy=fileName

[5] Hadoop: https://issues.apache.org/jira/browse/HADOOP-1911

TABLE VIII
COMPARATIVE ANALYSIS OF AAROHI

Research Solutions	Approach	Unsupervised	Lead Time (mins)	Test Time	Online	Target	Objective
Zheng et al. [28]	Genetic Algorithm (GA)	No	2 to 10	N/A	✓	BG/P	Failure Prediction
Hora [15]	ARIMA (Autoregression)	No	10	98 predictions/2 mins	✓	Netflix, non-HPC	Mem Leak/Node Crash
Fu et al. [14]	Episode Mining	No	N/A	N/A	×	Hadoop/LANL/BG/L	Root Cause Diagnosis
Berrocal et al. [13]	Void Search, PCA	No	N/A	4 secs/node	×	BG/Q	Fault Prediction
DeepLog [16]	LSTM	No	N/A	1.06 msecs/log entry	✓	OpenStack, BG/L	Anomaly Detection
CloudSeer [20]	Automatons, FSMs	N/A	N/A	2.36 msecs/log entry	✓	OpenStack	Anomaly Detection
Klinkenberg et al. [22]	Supervised Classifiers	No	17 & 22	N/A	×	HPC Cluster	Node Failures
Aarohi	Compiler-based	Yes	3	0.31 msecs/(length-18)	✓	Cray-HPC	Node Failures

TABLE IX
AAROHI ADAPTABILITY

#	HPC Systems		Distributed Systems (DS)	
	HPC5 (Cray-XK*)	HPC6 (IBM-BG/P)	Cassandra	Hadoop
P1	GPU* PMU communication error	MMCS detected error: power module	Unable to lock JVM memory	No node available for block
P2	L0 heartbeat fault	Network link errors detected	Server running in degraded mode	Could not obtain block*
P3	Voltage Fault Node	DDR correctable single symbol error(s)	Not starting RPC server as requested	DFS Read: java IOException*
P4	Machine Check Exception (MCE)	Kernel panic: soft-lockup: hung tasks	No host ID found	No live nodes contain current block
P5	Kernel Panic, Call Trace	Kill job * timed out	Exception in thread Thread*	DFSClient: Failed to connect
P6	GPU* memory page fault	Node System has halted	Exiting: error while processing commit log	NameNode: shutdown msg:

Moving away from the compute nodes can eliminate any possible impact on the job resource consumption. Second, if resources are scarce or load imbalance arises due to heavy workloads, the overall cluster computation has less impact.

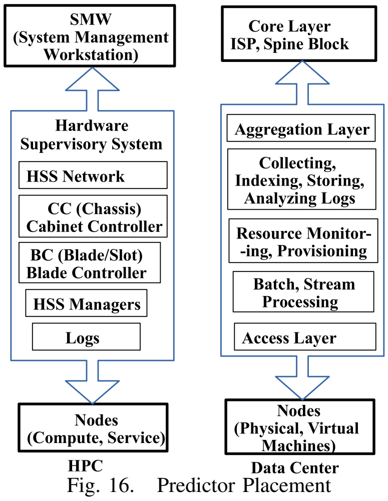
SMW
(System Management Workstation)
Hardware Supervisory System
HSS Network
CC (Chassis) Cabinet Controller
BC (Blade/Slot) Blade Controller
HSS Managers
Logs
Nodes (Compute, Service)
HPC
Core Layer
ISP, Spine Block
Aggregation Layer
Collecting, Indexing, Storing, Analyzing Logs
Resource Monitor- -ing, Provisioning
Batch, Stream Processing
Access Layer
Nodes (Physical, Virtual Machines)
Data Center

Fig. 16. Predictor Placement

In data centers, the aggregation layer is a centralized monitoring and processing layer, which performs batch or real-time streaming of data (e.g., Google Cloud Platform [29]). The data centers commonly assume a multi-tier model unlike the server-cluster model of HPC systems, making the design of a global predictor non-trivial. For a data center with 1000s of compute nodes, aggregating logs from all the hosts to a centralized location could throttle the network bandwidth, depending on the switch capacity and the network topology. Every physical host running diverse virtual machines and monitoring its own health can facilitate application-centric anomaly detection, but remains less effective for node failure prediction. Further deployment experiences, subject to future work, can unveil insights about addressing such practical concerns.

(2) Proactive Recovery Actions: We mentioned that enhancing inference speedup can aid proactive recovery actions such as quarantining unhealthy nodes [6] and cloning [24]. While Wang et al. [23] show live migration times <24 secs, Ouyang et al. [30] demonstrate that process migrations can be completed in 3.1 secs (10× speedup over conventional approaches). Adaptive lazy checkpointing [19] can also aid mitigation. Shutting down or preventing future job assignments onto flagged unhealthy nodes can prevent future failures. In <16 msecs prediction time, and >2 mins *effective* lead time, such proactive solutions become feasible in most cases.

V. RELATED WORK

Scientists have investigated anomalies in large-scale computing infrastructures across diverse research directions. HTM [31], Drain [32], [27], [33] and Spell [34] are parsers developed for online streaming logs. Recent DL model compression or parallel ML [35], [36] techniques are aimed at performance optimized training with large datasets as opposed to real-time testing of an incoming phrase. Besides, they may require specialized hardware or software support to reap parallelization benefits, incurring high computational costs. It is worthwhile to develop a simpler, more flexible scheme for clusters such as Aarohi. While [11]–[15], [17], [21], [22] have concerted efforts in predicting HPC failures, LogLens [16], [37], [38] have similar objectives in distributed systems. Whether offline or online, they either use ML-techniques [39] that are not scalable with a simpler failure model [40] or do not perform inference time analysis. REs, FSMs and CFGs have been used in the past [20], [41]–[43] for different contexts. However, prior work have not addressed generic machine translation of FCs. Moreover, their premise (application profiling, fault injection, source code reference) is considerably different from our chain-based failure prediction. Aarohi goes beyond these works by closing the manual translation gap between FC identification by ML and fast parser-based inferencing. Aarohi automatically generates lexing and parsing specifications for a language of FCs suitable for online prediction, which would also allow itself to be deployed in unsupervised dynamic re-training and re-generation of a new parser for enhanced FCs as they are being observed. Overall, Aarohi obtains prediction times low enough to provide effective lead times to failures, which is of primary concern.

VI. Conclusions

This paper proposes an online node failure predictor called Aarohi for HPC systems. The observed inference speedup is over 27.4× w.r.t. some known state-of-the-art approaches. This expeditious predictor provides runtime support for ML and obtains as high as 3 mins effective lead times considering prediction time. Aarohi demonstrates the feasibility of an auto-generated inference scheme based on parsing logs resulting in a speedup over prior methods. Additionally, it gives insights to log variations across systems and highlights the requirement of adaptability for sustainable prediction schemes.

Acknowledgment

The anonymous reviewers helped improve this paper. This work was supported in part by DOE subcontracts from Lawrence Berkeley and Lawrence Livermore National Labs, and NSF grants 1525609 and 0958311. This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[1] J. Brandt, A. Gentile, C. Martin, J. Repik, and N. Taerat, "New systems, new behaviors, new patterns: Monitoring insights from system standup," in *CLUSTER*. IEEE, 2015, pp. 658–665.

[2] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari, "Failures in large scale systems: long-term measurement, analysis, and implications," in *SC*. ACM, 2017, p. 44.

[3] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory errors in modern systems: The good, the bad, and the ugly," in *ASPLOS*, 2015, pp. 297–310.

[4] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith, "Unprotected computing: A large-scale study of dram raw error rate on a supercomputer," in *SC*. IEEE, 2016, pp. 645–655.

[5] G. Wang, L. Zhang, and W. Xu, "What can we learn from four years of data center hardware failures?" in *DSN*. IEEE, 2017, pp. 25–36.

[6] S. Gupta, D. Tiwari, C. Jantzi, J. H. Rogers, and D. Maxwell, "Understanding and exploiting spatial properties of system failures on extreme-scale HPC systems," in *DSN*. IEEE/IFIP, 2015, pp. 37–44.

[7] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux *et al.*, "Understanding gpu errors on large-scale hpc systems and the implications for system design and operation," in *HPCA*. IEEE, 2015, pp. 331–342.

[8] C. Di Martino, W. Kramer, Z. Kalbarczyk, and R. Iyer, "Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 hpc application runs," in *DSN*. IEEE, 2015, pp. 25–36.

[9] R. Birke, I. Giurgiu, L. Y. Chen, D. Wiesmann, and T. Engbersen, "Failure analysis of virtual and physical machines: patterns, causes and characteristics," in *DSN*. IEEE, 2014, pp. 1–12.

[10] N. El-Sayed, I. A. Stefanovici, G. Amvrosiadis, A. A. Hwang, and B. Schroeder, "Temperature management in data centers: why some (might) like it hot," *SIGMETRICS*, vol. 40, no. 1, pp. 163–174, 2012.

[11] S. Fu and C. Xu, "Exploring event correlation for failure prediction in coalitions of clusters," in *SC*. ACM/IEEE, 2007, p. 41.

[12] Z. Lan, Z. Zheng, and Y. Li, "Toward automated anomaly identification in large-scale systems," *TPDS*, vol. 21, no. 2, pp. 174–187, 2010.

[13] E. Berrocal, L. Yu, S. Wallace, M. E. Papka, and Z. Lan, "Exploring void search for fault detection on extreme scale systems," in *CLUSTER*. IEEE, 2014, pp. 1–9.

[14] X. Fu, R. Ren, S. A. McKee, J. Zhan, and N. Sun, "Digging deeper into cluster system logs for failure prediction and root cause diagnosis," in *CLUSTER*. IEEE, 2014, pp. 103–112.

[15] T. Pitakrat, D. Okanović, A. van Hoorn, and L. Grunske, "Hora: Architecture-aware online failure prediction," *Journal of Systems and Software*, 2017.

[16] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *CCS*. ACM, 2017, pp. 1285–1298.

[17] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. M. Brandt, V. J. Leung, M. Egele, and A. K. Coskun, "Diagnosing performance variations in HPC applications using machine learning," in *ISC*, 2017, pp. 355–373.

[18] F. Cappello, "Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities," *IJHCA*, vol. 23, no. 3, pp. 212–226, 2009.

[19] D. Tiwari, S. Gupta, and S. S. Vazhkudai, "Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems," in *DSN*. IEEE, 2014, pp. 25–36.

[20] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, "Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs," in *ASPLOS*, 2016, pp. 489–502.

[21] A. Gainaru, F. Cappello, M. Snir, and W. Kramer, "Failure prediction for HPC systems and applications: Current situation and open issues," *IJHPCA*, vol. 27, no. 3, pp. 273–282, 2013.

[22] J. Klinkenberg, C. Terboven, S. Lankes, and M. S. Müller, "Data mining-based analysis of HPC center operations," in *CLUSTER*. IEEE, 2017, pp. 766–773.

[23] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive process-level live migration in HPC environments," in *SC*. ACM/IEEE, 2008.

[24] A. Rezaei and F. Mueller, "DINO: divergent node cloning for sustained redundancy in HPC," in *CLUSTER*. IEEE, 2015, pp. 180–183.

[25] A. Das, F. Mueller, C. Siegel, and A. Vishnu, "Desh: Deep learning for system health prediction of lead times to failure in HPC," in *HPDC*. ACM, 2018.

[26] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, ser. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[27] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *ICSE (SEIP)*, 2019.

[28] Z. Zheng, Z. Lan, R. Gupta, S. Coghlan, and P. Beckman, "A practical failure prediction with location and lead time for blue gene/p," in *DSN-W*. IEEE, 2010, pp. 15–22.

[29] *Google Cloud Platform*. [Online]. Available: https://cloud.google.com/

[30] X. Ouyang, R. Rajachandrasekar, X. Besseron, and D. K. Panda, "High performance pipelined process migration with RDMA," in *CCGrid*. IEEE/ACM, 2011, pp. 314–323.

[31] S. Ahmad, A. Lavin, S. Purdy, and Z. Agha, "Unsupervised real-time anomaly detection for streaming data," *Neurocomputing*, vol. 262, pp. 134–147, 2017.

[32] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *ICWS*, 2017, pp. 33–40.

[33] N. Aussel, Y. Petetin, and S. Chabridon, "Improving performances of log mining for anomaly prediction through nlp-based log parsing," in *MASCOTS*. IEEE, 2018, pp. 237–243.

[34] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *ICDM*. IEEE, 2016, pp. 859–864.

[35] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *ICLR*, 2016.

[36] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for DNN training," in *ACM SOSP*, 2019, pp. 1–15.

[37] B. Debnath, M. Solaimani, M. A. Gulzar, N. Arora, C. Lumezanu, J. Xu, B. Zong, H. Zhang, G. Jiang, and L. Khan, "Loglens: A real-time log analysis system," in *ICDCS*. IEEE, 2018, pp. 1052–1062.

[38] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *ICDM*. IEEE, 2009, pp. 149–158.

[39] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods," *ACM Computing Surveys (CSUR)*, vol. 42, no. 3, p. 10, 2010.

[40] Y. Watanabe, H. Otsuka, M. Sonoda, S. Kikuchi, and Y. Matsumoto, "Online failure prediction in cloud datacenters by real-time message pattern learning," in *CloudCom*. IEEE, 2012, pp. 504–511.

[41] X. Yu and M. Becchi, "GPU acceleration of regular expression matching for large datasets: exploring the implementation space," in *CF*, 2013.

[42] M. Y. Chen, A. J. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer, "Path-based failure and evolution management," in *NSDI*, 2004, pp. 309–322.

[43] G. Bosman and S. Gruner, "Log file analysis with context-free grammars," in *IFIP Digital Forensics*, 2013, pp. 145–152.